```
},
"_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: ≥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: src/saaaaaa/core/orchestrator/base_executor_with_contract.py =====
from __future__ import annotations

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

from jsonschema import Draft7Validator

from saaaaaa.config.paths import PROJECT_ROOT
from saaaaaa.core.orchestrator.evidence_assembler import EvidenceAssembler
from saaaaaa.core.orchestrator.evidence_validator import EvidenceValidator
from saaaaaa.core.orchestrator.evidence_registry import get_global_registry

if TYPE_CHECKING:
    from saaaaaa.core.orchestrator.core import MethodExecutor, PreprocessedDocument
else:  # pragma: no cover - runtime avoids import to break cycles
    MethodExecutor = Any
    PreprocessedDocument = Any


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] = {}

    def __init__(
        self,
        method_executor: MethodExecutor,
        signal_registry: Any,
        config: Any,
        questionnaire_provider: Any,
        calibration_orchestrator: Any | None = None,
    ) -> None:
        try:
            from saaaaaa.core.orchestrator.core import MethodExecutor as _MethodExecutor
        except Exception as exc:  # pragma: no cover - defensive guard
            raise RuntimeError(
                "Failed to import MethodExecutor for BaseExecutorWithContract invariants.
"
                "Ensure saaaaaa.core.orchestrator.core is importable before constructing
contract executors."
            ) from exc
        if not isinstance(method_executor, _MethodExecutor):
            raise RuntimeError("A valid MethodExecutor instance is required for contract
executors.")
        self.method_executor = method_executor
        self.signal_registry = signal_registry
        self.config = config
        self.questionnaire_provider = questionnaire_provider
        self.calibration_orchestrator = calibration_orchestrator

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

    @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:
            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 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) -> dict[str, Any]:
        base_slot = cls.get_base_slot()
        if base_slot in cls._contract_cache:
            return cls._contract_cache[base_slot]

        # Try v3 contract first, then fall back to v2
        v3_path = PROJECT_ROOT / "config" / "executor_contracts" / f"{base_slot}.v3.json"
        v2_path = PROJECT_ROOT / "config" / "executor_contracts" / f"{base_slot}.json"

        if 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}. "
                f"Tried: {v3_path}, {v2_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[base_slot] = 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) → 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]):
        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}")

    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}"
            )

        contract = self._load_contract()
        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", [])

        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)

        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,
            "question_patterns": 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]))
        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}

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

            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", [])
    assembled = EvidenceAssembler.assemble(method_outputs, assembly_rules)
    evidence = assembled["evidence"]
    trace = assembled["trace"]

    validation_rules = contract.get("validation_rules", [])
    na_policy = contract.get("na_policy", "abort")
    validation_rules_object = {"rules": validation_rules, "na_policy": na_policy}
    validation = EvidenceValidator.validate(evidence, validation_rules_object)

    error_handling = contract.get("error_handling", {})
    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,
    }

    return result

  def _execute_v3(
    self,
    document: PreprocessedDocument,
    question_context_external: dict[str, Any],
    contract: dict[str, Any],
) -> dict[str, Any]:
    """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":
            abort_on_placeholder = self.config.get("abort_on_placeholder_calibration",
True) if hasattr(self.config, "get") else True
            if abort_on_placeholder:
                note = calibration.get("note", "No calibration note provided")
                raise RuntimeError(
                    f"Contract {base_slot} has placeholder calibration
(status={calibration_status}). "
                    f"Execution aborted per policy. Calibration note: {note}"
                )

        # 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)

        # 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]] = []

        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)

                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"
            )

        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

    # Evidence assembly
    evidence_assembly = contract["evidence_assembly"]
    assembly_rules = evidence_assembly["assembly_rules"]
    assembled = EvidenceAssembler.assemble(method_outputs, assembly_rules)
    evidence = assembled["evidence"]
    trace = assembled["trace"]

    # Validation with ENHANCED NA POLICY SUPPORT
    validation_rules_section = contract["validation_rules"]
    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}
    validation = EvidenceValidator.validate(evidence, validation_rules_object)

    # 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,
        }

        # Record evidence in global registry for provenance tracking
        registry = get_global_registry()
        registry.record_evidence(
            evidence_type="executor_result_v3",
            payload=result_data,

source_method=f"{self.__class__.__module__}.{self.__class__.__name__}.execute",
            question_id=question_id,
            document_id=getattr(document, "document_id", None),
        )

        # 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)

        # Generate human_readable_output if template exists
        human_readable_config = output_contract.get("human_readable_output", {})
        if human_readable_config:
            result_data["human_readable_output"] = self._generate_human_readable_output(
                evidence, validation, human_readable_config, contract
            )

        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:
            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)

===== FILE: src/saaaaaa/core/orchestrator/calibration_context.py =====
"""Calibration Context Module.

This module provides context-aware calibration capabilities for the orchestrator.
It defines context types, modifiers, and functions to adjust calibration parameters
based on question context, policy area, and unit of analysis.

Design Principles:
- Context is immutable and copied on modification
- Modifiers are composable and applied in sequence
- Context inference from question IDs is deterministic
- All adjustments are traceable and reversible
"""

from __future__ import annotations

import logging
import re
from dataclasses import dataclass, replace
from enum import Enum
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from .calibration_types import MethodCalibration

logger = logging.getLogger(__name__)


class PolicyArea(Enum):
    """Policy area classifications for context-aware calibration."""
    UNKNOWN = "unknown"
```

```
    FISCAL = "fiscal"
    SOCIAL = "social"
    INFRASTRUCTURE = "infrastructure"
    ENVIRONMENTAL = "environmental"
    GOVERNANCE = "governance"
    ECONOMIC = "economic"
    HEALTH = "health"
    EDUCATION = "education"
    SECURITY = "security"
    CULTURE = "culture"


class UnitOfAnalysis(Enum):
    """Unit of analysis for question context."""
    UNKNOWN = "unknown"
    BASELINE_GAP = "baseline_gap"
    INTERVENTION = "intervention"
    OUTCOME = "outcome"
    MECHANISM = "mechanism"
    CONTEXT = "context"
    TIMEFRAME = "timeframe"
    STAKEHOLDER = "stakeholder"
    RESOURCE = "resource"
    RISK = "risk"
    ASSUMPTION = "assumption"


@dataclass(frozen=True)
class CalibrationContext:
    """Context information for calibration adjustment.

    Attributes:
        question_id: Question identifier (e.g., "D1Q1")
        dimension: Dimension number (1-10)
        question_num: Question number within dimension
        policy_area: Policy area classification
        unit_of_analysis: Unit of analysis for the question
        method_position: Position of method in execution sequence (0-based)
        total_methods: Total number of methods to execute
    """
    question_id: str
    dimension: int = 0
    question_num: int = 0
    policy_area: PolicyArea = PolicyArea.UNKNOWN
    unit_of_analysis: UnitOfAnalysis = UnitOfAnalysis.UNKNOWN
    method_position: int = 0
    total_methods: int = 0

    @classmethod
    def from_question_id(cls, question_id: str) -> CalibrationContext:
        """Create context from question ID.

        Parses question IDs in format "D{dimension}Q{question}" (case-insensitive).
        Examples: "D1Q1", "d2q5", "D10Q25"

        Args:
            question_id: Question identifier string

        Returns:
            CalibrationContext with parsed dimension and question number
        """
        # Parse question ID format: D{dimension}Q{question}
        pattern = r"[dD](\d+)[qQ](\d+)"
        match = re.match(pattern, question_id)

        if match:
            dimension = int(match.group(1))
            question_num = int(match.group(2))
```

```
            return cls(
                question_id=question_id,
                dimension=dimension,
                question_num=question_num
            )
        else:
            logger.warning(f"Invalid question ID format: {question_id}")
            return cls(question_id=question_id, dimension=0, question_num=0)

    def with_policy_area(self, policy_area: PolicyArea) -> CalibrationContext:
        """Create a copy with updated policy area.

        Args:
            policy_area: New policy area

        Returns:
            New CalibrationContext with updated policy_area
        """
        return replace(self, policy_area=policy_area)

    def with_unit_of_analysis(self, unit_of_analysis: UnitOfAnalysis) ->
CalibrationContext:
        """Create a copy with updated unit of analysis.

        Args:
            unit_of_analysis: New unit of analysis

        Returns:
            New CalibrationContext with updated unit_of_analysis
        """
        return replace(self, unit_of_analysis=unit_of_analysis)

    def with_method_position(self, position: int, total: int) -> CalibrationContext:
        """Create a copy with updated method position.

        Args:
            position: Position in method execution sequence (0-based)
            total: Total number of methods

        Returns:
            New CalibrationContext with updated method_position and total_methods
        """
        return replace(self, method_position=position, total_methods=total)


@dataclass(frozen=True)
class CalibrationModifier:
    """Modifier for adjusting calibration parameters based on context.

    All multipliers default to 1.0 (no change). Values outside valid ranges
    are clamped during application.

    Attributes:
        min_evidence_multiplier: Multiplier for min_evidence_snippets
        max_evidence_multiplier: Multiplier for max_evidence_snippets
        contradiction_tolerance_multiplier: Multiplier for contradiction_tolerance
        uncertainty_penalty_multiplier: Multiplier for uncertainty_penalty
        aggregation_weight_multiplier: Multiplier for aggregation_weight
        sensitivity_multiplier: Multiplier for sensitivity
    """
    min_evidence_multiplier: float = 1.0
    max_evidence_multiplier: float = 1.0
    contradiction_tolerance_multiplier: float = 1.0
    uncertainty_penalty_multiplier: float = 1.0
    aggregation_weight_multiplier: float = 1.0
    sensitivity_multiplier: float = 1.0

    def apply(self, calibration: MethodCalibration) -> MethodCalibration:
```

```
        """Apply modifier to a calibration.

        Args:
            calibration: Base MethodCalibration to modify

        Returns:
            New MethodCalibration with adjusted parameters
        """
        # Import at runtime to avoid circular dependency at module load time
        from .calibration_types import MethodCalibration

        # Apply multipliers and clamp to valid ranges
        min_evidence = int(calibration.min_evidence_snippets *
self.min_evidence_multiplier)
        max_evidence = int(calibration.max_evidence_snippets *
self.max_evidence_multiplier)

        # Ensure min <= max
        if min_evidence > max_evidence:
            min_evidence, max_evidence = max_evidence, min_evidence

        # Clamp evidence counts to reasonable ranges
        min_evidence = max(1, min_evidence)
        max_evidence = max(min_evidence, min(100, max_evidence))

        # Apply multipliers and clamp to [0.0, 1.0]
        contradiction_tolerance = max(0.0, min(1.0,
            calibration.contradiction_tolerance * self.contradiction_tolerance_multiplier
        ))

        uncertainty_penalty = max(0.0, min(1.0,
            calibration.uncertainty_penalty * self.uncertainty_penalty_multiplier
        ))

        aggregation_weight = max(0.0,
            calibration.aggregation_weight * self.aggregation_weight_multiplier
        )

        sensitivity = max(0.0, min(1.0,
            calibration.sensitivity * self.sensitivity_multiplier
        ))

        return MethodCalibration(
            score_min=calibration.score_min,
            score_max=calibration.score_max,
            min_evidence_snippets=min_evidence,
            max_evidence_snippets=max_evidence,
            contradiction_tolerance=contradiction_tolerance,
            uncertainty_penalty=uncertainty_penalty,
            aggregation_weight=aggregation_weight,
            sensitivity=sensitivity,
            requires_numeric_support=calibration.requires_numeric_support,
            requires_temporal_support=calibration.requires_temporal_support,
            requires_source_provenance=calibration.requires_source_provenance,
        )


# Dimension-specific modifiers
_DIMENSION_MODIFIERS = {
    1: CalibrationModifier(min_evidence_multiplier=1.3, sensitivity_multiplier=1.1),
    2: CalibrationModifier(max_evidence_multiplier=1.2,
contradiction_tolerance_multiplier=0.8),
    3: CalibrationModifier(min_evidence_multiplier=1.2,
uncertainty_penalty_multiplier=0.9),
    4: CalibrationModifier(sensitivity_multiplier=1.2),
    5: CalibrationModifier(min_evidence_multiplier=1.1, max_evidence_multiplier=1.1),
    6: CalibrationModifier(contradiction_tolerance_multiplier=0.9),
    7: CalibrationModifier(uncertainty_penalty_multiplier=0.85),
```

```
    8: CalibrationModifier(aggregation_weight_multiplier=1.15),
    9: CalibrationModifier(sensitivity_multiplier=1.15),
    10: CalibrationModifier(min_evidence_multiplier=1.4, sensitivity_multiplier=1.2),
}

# Policy area modifiers
_POLICY_AREA_MODIFIERS = {
    PolicyArea.FISCAL: CalibrationModifier(
        min_evidence_multiplier=1.3,
        sensitivity_multiplier=1.1
    ),
    PolicyArea.SOCIAL: CalibrationModifier(
        max_evidence_multiplier=1.2,
        uncertainty_penalty_multiplier=0.9
    ),
    PolicyArea.INFRASTRUCTURE: CalibrationModifier(
        contradiction_tolerance_multiplier=0.8,
        sensitivity_multiplier=1.1
    ),
    PolicyArea.ENVIRONMENTAL: CalibrationModifier(
        min_evidence_multiplier=1.2,
        uncertainty_penalty_multiplier=0.85
    ),
}

# Unit of analysis modifiers
_UNIT_OF_ANALYSIS_MODIFIERS = {
    UnitOfAnalysis.BASELINE_GAP: CalibrationModifier(
        min_evidence_multiplier=1.4,
        sensitivity_multiplier=1.2
    ),
    UnitOfAnalysis.INTERVENTION: CalibrationModifier(
        contradiction_tolerance_multiplier=0.9,
        sensitivity_multiplier=1.1
    ),
    UnitOfAnalysis.OUTCOME: CalibrationModifier(
        min_evidence_multiplier=1.3,
        uncertainty_penalty_multiplier=0.8
    ),
    UnitOfAnalysis.MECHANISM: CalibrationModifier(
        max_evidence_multiplier=1.2,
        sensitivity_multiplier=1.15
    ),
}


def resolve_contextual_calibration(
    base_calibration: MethodCalibration,
    context: CalibrationContext | None = None
) -> MethodCalibration:
    """Resolve calibration with context-aware adjustments.

    Applies modifiers based on:
    1. Dimension (if context.dimension > 0)
    2. Policy area (if not UNKNOWN)
    3. Unit of analysis (if not UNKNOWN)

    Args:
        base_calibration: Base MethodCalibration
        context: Optional CalibrationContext with adjustment information

    Returns:
        Adjusted MethodCalibration
    """
    if context is None:
        return base_calibration

    result = base_calibration
```

```
    # Apply dimension modifier
    if context.dimension > 0 and context.dimension in _DIMENSION_MODIFIERS:
        modifier = _DIMENSION_MODIFIERS[context.dimension]
        result = modifier.apply(result)
        logger.debug(f"Applied dimension {context.dimension} modifier")

    # Apply policy area modifier
    if context.policy_area != PolicyArea.UNKNOWN:
        if context.policy_area in _POLICY_AREA_MODIFIERS:
            modifier = _POLICY_AREA_MODIFIERS[context.policy_area]
            result = modifier.apply(result)
            logger.debug(f"Applied policy area {context.policy_area.value} modifier")

    # Apply unit of analysis modifier
    if context.unit_of_analysis != UnitOfAnalysis.UNKNOWN:
        if context.unit_of_analysis in _UNIT_OF_ANALYSIS_MODIFIERS:
            modifier = _UNIT_OF_ANALYSIS_MODIFIERS[context.unit_of_analysis]
            result = modifier.apply(result)
            logger.debug(f"Applied unit of analysis {context.unit_of_analysis.value}
modifier")

    return result


def infer_context_from_question_id(question_id: str) -> CalibrationContext:
    """Infer context from question ID.

    This is a convenience function that creates a CalibrationContext from
    a question ID. Additional context can be added using the with_* methods.

    Args:
        question_id: Question identifier (e.g., "D1Q1")

    Returns:
        CalibrationContext with inferred dimension and question number
    """
    return CalibrationContext.from_question_id(question_id)


__all__ = [
    "CalibrationContext",
    "CalibrationModifier",
    "PolicyArea",
    "UnitOfAnalysis",
    "resolve_contextual_calibration",
    "infer_context_from_question_id",
]

===== FILE: src/saaaaaa/core/orchestrator/calibration_registry.py =====
"""Calibration Registry Module.

This module provides base calibration resolution for orchestrator methods.
It defines the MethodCalibration dataclass and functions to resolve calibration
parameters for methods, with optional context-aware adjustments.

Design Principles:
- Base calibration is context-independent
- Reads from config/intrinsic_calibration.json
- Provides fallback defaults for uncalibrated methods
- Supports context-aware resolution via calibration_context module
"""

from __future__ import annotations

import json
import logging
from dataclasses import dataclass
```

```
from pathlib import Path
from typing import Any

from ..calibration.intrinsic_loader import IntrinsicScoreLoader

logger = logging.getLogger(__name__)

# Canonical repository root
# Path hierarchy: calibration_registry.py -> orchestrator -> core -> saaaaaa -> src ->
REPO_ROOT
_REPO_ROOT = Path(__file__).resolve().parents[4]
_CALIBRATION_FILE = _REPO_ROOT / "config" / "intrinsic_calibration.json"


from .calibration_types import MethodCalibration


# Cache for loaded calibration data
_calibration_cache: dict[str, Any] | None = None


def _load_calibration_data() -> dict[str, Any]:
    """Load calibration data from config file.

    Returns:
        Dictionary containing calibration data
    """
    global _calibration_cache

    if _calibration_cache is not None:
        return _calibration_cache

    if not _CALIBRATION_FILE.exists():
        logger.warning(f"Calibration file not found: {_CALIBRATION_FILE}")
        _calibration_cache = {}
        return _calibration_cache

    try:
        with open(_CALIBRATION_FILE, encoding='utf-8') as f:
            data = json.load(f)
            _calibration_cache = data
            logger.info(f"Loaded calibration data from {_CALIBRATION_FILE}")
            return data
    except Exception as e:
        logger.error(f"Failed to load calibration data: {e}")
        _calibration_cache = {}
        return _calibration_cache


def _get_default_calibration() -> MethodCalibration:
    """Get default calibration for uncalibrated methods.

    Returns:
        Default MethodCalibration with conservative parameters
    """
    return MethodCalibration(
        score_min=0.0,
        score_max=1.0,
        min_evidence_snippets=3,
        max_evidence_snippets=15,
        contradiction_tolerance=0.1,
        uncertainty_penalty=0.3,
        aggregation_weight=1.0,
        sensitivity=0.75,
        requires_numeric_support=False,
        requires_temporal_support=False,
        requires_source_provenance=True,
    )
```

```
def resolve_calibration(class_name: str, method_name: str) -> MethodCalibration:
    """Resolve base calibration for a method.

    This function looks up calibration parameters from the intrinsic calibration
    file. If no calibration is found, it returns conservative defaults.

    Args:
        class_name: Name of the class (e.g., "SemanticAnalyzer")
        method_name: Name of the method (e.g., "extract_entities")

    Returns:
        MethodCalibration with parameters for this method
    """
    data = _load_calibration_data()

    # Try to find calibration for this specific method
    method_key = f"{class_name}.{method_name}"

    # Check if calibration exists for this method
    if method_key in data:
        method_data = data[method_key]
        try:
            return MethodCalibration(
                score_min=method_data.get("score_min", 0.0),
                score_max=method_data.get("score_max", 1.0),
                min_evidence_snippets=method_data.get("min_evidence_snippets", 3),
                max_evidence_snippets=method_data.get("max_evidence_snippets", 15),
                contradiction_tolerance=method_data.get("contradiction_tolerance", 0.1),
                uncertainty_penalty=method_data.get("uncertainty_penalty", 0.3),
                aggregation_weight=method_data.get("aggregation_weight", 1.0),
                sensitivity=method_data.get("sensitivity", 0.75),
                requires_numeric_support=method_data.get("requires_numeric_support",
False),
                requires_temporal_support=method_data.get("requires_temporal_support",
False),
                requires_source_provenance=method_data.get("requires_source_provenance",
True),
            )
        except (KeyError, ValueError) as e:
            logger.warning(f"Invalid calibration for {method_key}: {e}. Using defaults.")
            return _get_default_calibration()

    # No specific calibration found, use defaults
    logger.debug(f"No calibration found for {method_key}, using defaults")
    return _get_default_calibration()


def resolve_calibration_with_context(
    class_name: str,
    method_name: str,
    question_id: str | None = None,
    **kwargs: Any
) -> MethodCalibration:
    """Resolve calibration with context-aware adjustments.

    This function first resolves base calibration, then applies context-specific
    modifiers based on the question ID and other context information.

    Args:
        class_name: Name of the class
        method_name: Name of the method
        question_id: Question ID for context inference (e.g., "D1Q1")
        **kwargs: Additional context parameters (policy_area, unit_of_analysis, etc.)

    Returns:
        MethodCalibration with context-aware adjustments applied
```

```
    """
    # Get base calibration
    base_calibration = resolve_calibration(class_name, method_name)

    # If no question_id provided, return base calibration
    if question_id is None:
        return base_calibration

    # Import context module to avoid circular dependency
    try:
        from .calibration_context import (
            CalibrationContext,
            resolve_contextual_calibration,
        )

        # Create context from question ID
        context = CalibrationContext.from_question_id(question_id)

        # Apply any additional context from kwargs
        if "policy_area" in kwargs:
            context = context.with_policy_area(kwargs["policy_area"])
        if "unit_of_analysis" in kwargs:
            context = context.with_unit_of_analysis(kwargs["unit_of_analysis"])
        if "method_position" in kwargs and "total_methods" in kwargs:
            context = context.with_method_position(
                kwargs["method_position"],
                kwargs["total_methods"]
            )

        # Apply contextual adjustments
        return resolve_contextual_calibration(base_calibration, context)

    except ImportError as e:
        logger.warning(f"Context module not available: {e}. Using base calibration.")
        return base_calibration


def get_calibration_manifest_data(
    calibration_path: str | Path = "config/intrinsic_calibration.json",
    method_ids: list[str] | None = None,
) -> dict[str, Any]:
    """
    Return canonical calibration manifest data for verification.

    Args:
        calibration_path: Path to intrinsic calibration JSON.
        method_ids: Optional method identifiers to include in detail.

    Returns:
        Dictionary with version, hash, statistics, and optional per-method data.
    """
    loader = IntrinsicScoreLoader(calibration_path)
    return loader.get_manifest_snapshot(method_ids)


__all__ = [
    "MethodCalibration",
    "resolve_calibration",
    "resolve_calibration_with_context",
    "get_calibration_manifest_data",
]

===== FILE: src/saaaaaa/core/orchestrator/calibration_types.py =====
"""Calibration Types Module.

This module defines the core data structures for calibration to avoid circular
dependencies.
"""
```

```
from __future__ import annotations

from dataclasses import dataclass

@dataclass(frozen=True)
class MethodCalibration:
    """Calibration parameters for an orchestrator method.

    Attributes:
        score_min: Minimum score value (typically 0.0)
        score_max: Maximum score value (typically 1.0)
        min_evidence_snippets: Minimum number of evidence snippets required
        max_evidence_snippets: Maximum number of evidence snippets to collect
        contradiction_tolerance: Tolerance for contradictory evidence (0.0-1.0)
        uncertainty_penalty: Penalty for uncertain evidence (0.0-1.0)
        aggregation_weight: Weight in aggregation (typically 1.0)
        sensitivity: Method sensitivity to input variations (0.0-1.0)
        requires_numeric_support: Whether method requires numeric evidence
        requires_temporal_support: Whether method requires temporal evidence
        requires_source_provenance: Whether method requires source provenance
    """
    score_min: float
    score_max: float
    min_evidence_snippets: int
    max_evidence_snippets: int
    contradiction_tolerance: float
    uncertainty_penalty: float
    aggregation_weight: float
    sensitivity: float
    requires_numeric_support: bool
    requires_temporal_support: bool
    requires_source_provenance: bool

    def __post_init__(self):
        """Validate calibration parameters."""
        if not 0.0 <= self.score_min <= self.score_max <= 1.0:
            raise ValueError(f"Invalid score range: [{self.score_min}, {self.score_max}]")
        if not 0 <= self.min_evidence_snippets <= self.max_evidence_snippets:
            raise ValueError(
                f"Invalid evidence range: [{self.min_evidence_snippets},
{self.max_evidence_snippets}]"
            )
        if not 0.0 <= self.contradiction_tolerance <= 1.0:
            raise ValueError(f"Invalid contradiction_tolerance:
{self.contradiction_tolerance}")
        if not 0.0 <= self.uncertainty_penalty <= 1.0:
            raise ValueError(f"Invalid uncertainty_penalty: {self.uncertainty_penalty}")
        if not 0.0 <= self.sensitivity <= 1.0:
            raise ValueError(f"Invalid sensitivity: {self.sensitivity}")

===== FILE: src/saaaaaa/core/orchestrator/catalogo_completo_canonico.py =====
"""
Canonical Method Catalog - AUTO-GENERATED

This module provides the canonical, authoritative registry of all methods
in the policy analysis system. It is derived from:
    config/rules/METODOS/catalogo_completo_canonico.json

STRICT RULES:
1. This is the SINGLE SOURCE OF TRUTH for method identifiers and signatures
2. NO modifications without updating the source JSON
3. Local usage that conflicts with this catalog is WRONG
4. All aliases, misspellings, and variants must be normalized to canonical forms

Generated from catalog version: 3.0.0
Total canonical methods: 593
"""
```

```
import json
from dataclasses import dataclass
from enum import Enum
from pathlib import Path


class MethodComplexity(Enum):
    """Canonical complexity levels"""
    LOW = "LOW"
    MEDIUM = "MEDIUM"
    HIGH = "HIGH"
    UNKNOWN = "UNKNOWN"


class MethodPriority(Enum):
    """Canonical priority levels"""
    LOW = "LOW"
    MEDIUM = "MEDIUM"
    HIGH = "HIGH"
    CRITICAL = "CRITICAL"
    UNKNOWN = "UNKNOWN"


@dataclass(frozen=True)
class ExecutionRequirements:
    """Execution requirements for a method"""
    computational: str  # LOW, MEDIUM, HIGH
    memory: str  # LOW, MEDIUM, HIGH
    io_bound: bool
    stateful: bool


@dataclass(frozen=True)
class CanonicalMethod:
    """
    Canonical method definition.

    This is the authoritative definition of a method in the system.
    All references to this method MUST use these exact identifiers.
    """
    class_name: str
    method_name: str
    file: str
    signature: str
    complexity: MethodComplexity
    priority: MethodPriority
    line_number: int
    aptitude_score: float
    execution_requirements: ExecutionRequirements
    dependencies: list[str]
    prerequisites: list[str]
    risks: list[str]
    docstring: str
    decorators: list[str]

    @property
    def fqn(self) -> str:
        """Fully qualified name: ClassName.method_name"""
        return f"{self.class_name}.{self.method_name}"

    @property
    def catalog_key(self) -> tuple:
        """Canonical tuple key for lookups"""
        return (self.class_name, self.method_name)


class CanonicalMethodCatalog:
```

```
    """
    The canonical method catalog.

    This class provides programmatic access to the authoritative method registry.
    It enforces strict canonicalization and rejects any undefined methods.
    """

    def __init__(self) -> None:
        self._methods: dict[tuple, CanonicalMethod] = {}
        self._by_class: dict[str, list[CanonicalMethod]] = {}
        self._by_file: dict[str, list[CanonicalMethod]] = {}
        self._metadata: dict = {}
        self._summary: dict = {}
        self._load_catalog()

    def find_repo_root(self, start_path: Path) -> Path:
        """Find the repository root by looking for .git or config directory"""
        current = start_path.resolve()
        while current != current.parent:
            if (current / ".git").exists() or (current / "config").exists():
                return current
            current = current.parent
        raise FileNotFoundError("Could not locate repository root")

    def _load_catalog(self) -> None:
        """Load the canonical catalog from JSON"""
        repo_root = self.find_repo_root(Path(__file__))
        catalog_path = repo_root / "config" / "rules" / "METODOS" /
"catalogo_completo_canonico.json"

        if not catalog_path.exists():
            raise FileNotFoundError(
                f"Canonical catalog not found at {catalog_path}. "
                "Cannot proceed without the authoritative method registry."
            )

        with open(catalog_path, encoding='utf-8') as f:
            data = json.load(f)

        self._metadata = data.get('metadata', {})
        self._summary = data.get('summary', {})

        # Build method registry
        for file_name, file_data in data.get('files', {}).items():
            for method_data in file_data.get('methods', []):
                method = self._parse_method(method_data, file_name)

                # Register by canonical key
                # Note: catalog may have duplicates from different files
                # Use the first occurrence as canonical
                key = method.catalog_key
                if key not in self._methods:
                    self._methods[key] = method

                # Index by class
                if method.class_name not in self._by_class:
                    self._by_class[method.class_name] = []
                self._by_class[method.class_name].append(method)

                # Index by file
                if file_name not in self._by_file:
                    self._by_file[file_name] = []
                self._by_file[file_name].append(method)

    def _parse_method(self, method_data: dict, file_name: str) -> CanonicalMethod:
        """Parse method data into CanonicalMethod"""
        exec_req_data = method_data.get('execution_requirements', {})
        exec_req = ExecutionRequirements(
```

```
            computational=exec_req_data.get('computational', 'UNKNOWN'),
            memory=exec_req_data.get('memory', 'UNKNOWN'),
            io_bound=exec_req_data.get('io_bound', False),
            stateful=exec_req_data.get('stateful', False),
        )

        try:
            complexity = MethodComplexity(method_data.get('complexity', 'UNKNOWN'))
        except ValueError:
            complexity = MethodComplexity.UNKNOWN

        try:
            priority = MethodPriority(method_data.get('priority', 'UNKNOWN'))
        except ValueError:
            priority = MethodPriority.UNKNOWN

        return CanonicalMethod(
            class_name=method_data.get('class', ''),
            method_name=method_data.get('method_name', ''),
            file=file_name,
            signature=method_data.get('signature', ''),
            complexity=complexity,
            priority=priority,
            line_number=method_data.get('line_number', 0),
            aptitude_score=method_data.get('aptitude_score', 0.0),
            execution_requirements=exec_req,
            dependencies=method_data.get('dependencies', []),
            prerequisites=method_data.get('prerequisites', []),
            risks=method_data.get('risks', []),
            docstring=method_data.get('docstring', 'No documentation available'),
            decorators=method_data.get('decorators', []),
        )

    def get_method(self, class_name: str, method_name: str) -> CanonicalMethod | None:
        """
        Retrieve a canonical method definition.

        Args:
            class_name: Exact class name
            method_name: Exact method name

        Returns:
            CanonicalMethod if found, None otherwise
        """
        return self._methods.get((class_name, method_name))

    def is_canonical(self, class_name: str, method_name: str) -> bool:
        """Check if a method is in the canonical catalog"""
        return (class_name, method_name) in self._methods

    def get_methods_by_class(self, class_name: str) -> list[CanonicalMethod]:
        """Get all canonical methods for a class"""
        return self._by_class.get(class_name, [])

    def get_methods_by_file(self, file_name: str) -> list[CanonicalMethod]:
        """Get all canonical methods from a file"""
        return self._by_file.get(file_name, [])

    def all_methods(self) -> list[CanonicalMethod]:
        """Return all canonical methods"""
        return list(self._methods.values())

    def all_classes(self) -> set[str]:
        """Return all canonical class names"""
        return set(self._by_class.keys())

    def all_files(self) -> set[str]:
        """Return all canonical file names"""
```

```
        return set(self._by_file.keys())

    @property
    def total_methods(self) -> int:
        """Total number of canonical methods"""
        return len(self._methods)

    @property
    def catalog_version(self) -> str:
        """Catalog version"""
        return self._metadata.get('version', 'unknown')

    @property
    def generated_at(self) -> str:
        """Catalog generation timestamp"""
        return self._metadata.get('generated_at', 'unknown')

    def validate_method_reference(self, class_name: str, method_name: str) -> bool:
        """
        Validate that a method reference matches the canonical catalog.

        Raises:
            ValueError: If method is not in canonical catalog

        Returns:
            True if valid
        """
        if not self.is_canonical(class_name, method_name):
            raise ValueError(
                f"Method {class_name}.{method_name} is NOT in the canonical catalog. "
                f"This is a DEFECT. Either:\n"
                f"  1. The method name/class is misspelled (fix the reference)\n"
                f"  2. The method is new (add to catalog first)\n"
                f"  3. The catalog is outdated (regenerate it)\n"
                f"Canonical catalog has {self.total_methods} methods. "
                f"Use CATALOG.all_classes() to see available classes."
            )
        return True

    def get_summary_stats(self) -> dict:
        """Get summary statistics about the catalog"""
        return {
            "total_methods": self.total_methods,
            "total_classes": len(self.all_classes()),
            "total_files": len(self.all_files()),
            "by_complexity": self._summary.get('by_complexity', {}),
            "by_priority": self._summary.get('by_priority', {}),
            "version": self.catalog_version,
            "generated_at": self.generated_at,
        }


# Global singleton instance
CATALOG = CanonicalMethodCatalog()


def get_canonical_method(class_name: str, method_name: str) -> CanonicalMethod | None:
    """
    Get a canonical method definition.

    This is the primary entry point for method lookups.

    Args:
        class_name: Exact canonical class name
        method_name: Exact canonical method name

    Returns:
        CanonicalMethod if found, None otherwise
```

```
    """
    return CATALOG.get_method(class_name, method_name)


def validate_method_is_canonical(class_name: str, method_name: str) -> bool:
    """
    Validate that a method is in the canonical catalog.

    Raises ValueError if not found.
    """
    return CATALOG.validate_method_reference(class_name, method_name)


def get_all_canonical_methods() -> list[CanonicalMethod]:
    """Get all canonical methods"""
    return CATALOG.all_methods()


def get_catalog_summary() -> dict:
    """Get catalog summary statistics"""
    return CATALOG.get_summary_stats()

===== FILE: src/saaaaaa/core/orchestrator/chunk_router.py =====
"""
Chunk Router for SPC Exploitation.

Routes semantic chunks to appropriate executors based on chunk type,
enabling targeted execution and reducing redundant processing.
"""

from __future__ import annotations

from dataclasses import dataclass
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from .core import ChunkData


@dataclass
class ChunkRoute:
    """Routing decision for a single chunk."""
    chunk_id: int
    chunk_type: str
    executor_class: str
    methods: list[tuple[str, str]]  # [(class_name, method_name), ...]
    skip_reason: str | None = None


class ChunkRouter:
    """
    Routes chunks to appropriate executors based on semantic type.

    This enables chunk-aware execution, where different chunk types
    are processed by the most relevant executors, avoiding unnecessary
    full-document processing.
    """

    # TYPE-TO-EXECUTOR MAPPING
    # Maps chunk types to executor base slots (e.g., "D1Q1", "D2Q3")
    ROUTING_TABLE: dict[str, list[str]] = {
        "diagnostic": ["D1Q1", "D1Q2", "D1Q5"],  # Baseline/gap analysis executors
        "activity": ["D2Q1", "D2Q2", "D2Q3", "D2Q4", "D2Q5"],  # Activity/intervention
executors
        "indicator": ["D3Q1", "D3Q2", "D4Q1", "D5Q1"],  # Metric/indicator executors
        "resource": ["D1Q3", "D2Q4", "D5Q5"],  # Financial/resource executors
        "temporal": ["D1Q5", "D3Q4", "D5Q4"],  # Timeline/temporal executors
        "entity": ["D2Q3", "D3Q3"],  # Responsibility/entity executors
```

```
}

# METHODS THAT MUST SEE FULL GRAPH
# These methods require access to the complete chunk graph
GRAPH_METHODS: set[str] = {
    "TeoriaCambio.construir_grafo_causal",
    "CausalExtractor.extract_causal_hierarchy",
    "AdvancedDAGValidator.calculate_acyclicity_pvalue",
    "CrossReferenceValidator.validate_internal_consistency",
}

def route_chunk(self, chunk: ChunkData) -> ChunkRoute:
    """
    Determine executor routing for a chunk.

    Args:
        chunk: ChunkData to route

    Returns:
        ChunkRoute with executor assignment and method list
    """
    executor_classes = self.ROUTING_TABLE.get(chunk.chunk_type, [])

    if not executor_classes:
        return ChunkRoute(
            chunk_id=chunk.id,
            chunk_type=chunk.chunk_type,
            executor_class="",
            methods=[],
            skip_reason=f"No executor mapping for chunk type '{chunk.chunk_type}'"
        )

    # Get primary executor for this chunk type
    primary_executor = executor_classes[0]

    # Get method subset for this chunk type
    # Note: Actual method filtering would require loading executor configs
    # For now, we return empty list and let execute_chunk filter
    methods: list[tuple[str, str]] = []

    return ChunkRoute(
        chunk_id=chunk.id,
        chunk_type=chunk.chunk_type,
        executor_class=primary_executor,
        methods=methods,
    )

def should_use_full_graph(self, method_name: str, class_name: str = "") -> bool:
    """
    Check if a method requires access to the full chunk graph.

    Args:
        method_name: Name of the method
        class_name: Optional class name

    Returns:
        True if method needs full graph access
    """
    full_name = f"{class_name}.{method_name}" if class_name else method_name
    return full_name in self.GRAPH_METHODS or method_name in self.GRAPH_METHODS

def get_relevant_executors(self, chunk_type: str) -> list[str]:
    """
    Get list of executors relevant to a chunk type.

    Args:
        chunk_type: Type of chunk
```

```
        Returns:
            List of executor base slots
        """
        return self.ROUTING_TABLE.get(chunk_type, [])

===== FILE: src/saaaaaa/core/orchestrator/class_registry.py =====
"""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.
_CLASS_PATHS: Mapping[str, str] = {
    "IndustrialPolicyProcessor":
"saaaaaa.processing.policy_processor.IndustrialPolicyProcessor",
    "PolicyTextProcessor": "saaaaaa.processing.policy_processor.PolicyTextProcessor",
    "BayesianEvidenceScorer":
"saaaaaa.processing.policy_processor.BayesianEvidenceScorer",
    "PolicyContradictionDetector":
"saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetector",
    "TemporalLogicVerifier":
"saaaaaa.analysis.contradiction_deteccion.TemporalLogicVerifier",
    "BayesianConfidenceCalculator":
"saaaaaa.analysis.contradiction_deteccion.BayesianConfidenceCalculator",
    "PDETMunicipalPlanAnalyzer":
"saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer",
    "CDAFFramework": "saaaaaa.analysis.derek_beach.CDAFFramework",
    "CausalExtractor": "saaaaaa.analysis.derek_beach.CausalExtractor",
    "OperationalizationAuditor": "saaaaaa.analysis.derek_beach.OperationalizationAuditor",
    "FinancialAuditor": "saaaaaa.analysis.derek_beach.FinancialAuditor",
    "BayesianMechanismInference":
"saaaaaa.analysis.derek_beach.BayesianMechanismInference",
    "BayesianNumericalAnalyzer":
"saaaaaa.processing.embedding_policy.BayesianNumericalAnalyzer",
    "PolicyAnalysisEmbedder":
"saaaaaa.processing.embedding_policy.PolicyAnalysisEmbedder",
    "AdvancedSemanticChunker":
"saaaaaa.processing.embedding_policy.AdvancedSemanticChunker",
    # SemanticChunker is an alias maintained for backwards compatibility.
    "SemanticChunker": "saaaaaa.processing.embedding_policy.AdvancedSemanticChunker",
    "SemanticAnalyzer": "saaaaaa.analysis.Analyzer_one.SemanticAnalyzer",
    "PerformanceAnalyzer": "saaaaaa.analysis.Analyzer_one.PerformanceAnalyzer",
    "TextMiningEngine": "saaaaaa.analysis.Analyzer_one.TextMiningEngine",
    "MunicipalOntology": "saaaaaa.analysis.Analyzer_one.MunicipalOntology",
    "TeoriaCambio": "saaaaaa.analysis.teoria_cambio.TeoriaCambio",
    "AdvancedDAGValidator": "saaaaaa.analysis.teoria_cambio.AdvancedDAGValidator",
    "D1_Q1_QuantitativeBaselineExtractor":
"saaaaaa.core.orchestrator.executors.D1_Q1_QuantitativeBaselineExtractor",
    "D1_Q2_ProblemDimensioningAnalyzer":
"saaaaaa.core.orchestrator.executors.D1_Q2_ProblemDimensioningAnalyzer",
    "SemanticProcessor": "saaaaaa.processing.semantic_chunking_policy.SemanticProcessor",
    "BayesianCounterfactualAuditor":
"saaaaaa.analysis.derek_beach.BayesianCounterfactualAuditor",
}

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: src/saaaaaa/core/orchestrator/contract_loader.py =====
"""
Legacy contract loader shim.

This module exists only to satisfy historical imports. The canonical executor
contract loading path is implemented in BaseExecutorWithContract._load_contract.
Use that path instead of this loader for all new code.
"""

from __future__ import annotations

from dataclasses import dataclass
from typing import Any
```

```
class LoadError(RuntimeError):
    """Raised when legacy contract loading is invoked."""


@dataclass
class LoadResult:
    """Placeholder result for legacy contract loading."""

    contracts: dict[str, Any] | None = None
    errors: list[str] | None = None


class JSONContractLoader:
    """Legacy loader shim that fails fast.

    Contract executors should load contracts via BaseExecutorWithContract._load_contract,
    which validates against the canonical schema. This shim prevents silent fallbacks.
    """

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

    def load_file(self, *args: Any, **kwargs: Any) -> LoadResult:
        raise LoadError(
            "JSONContractLoader is obsolete. Use BaseExecutorWithContract._load_contract "
            "for executor contracts."
        )

    def load_directory(self, *args: Any, **kwargs: Any) -> LoadResult:
        raise LoadError(
            "JSONContractLoader is obsolete. Use BaseExecutorWithContract._load_contract "
            "for executor contracts."
        )

    def load_multiple(self, *args: Any, **kwargs: Any) -> LoadResult:
        raise LoadError(
            "JSONContractLoader is obsolete. Use BaseExecutorWithContract._load_contract "
            "for executor contracts."
        )


__all__ = ["JSONContractLoader", "LoadError", "LoadResult"]

===== FILE: src/saaaaaa/core/orchestrator/core.py =====
"""Core orchestrator classes, data models, and execution engine.

This module contains the fundamental building blocks for orchestration:
- Data models (PreprocessedDocument, Evidence, PhaseResult, etc.)
- Abort signaling (AbortSignal, AbortRequested)
- Resource management (ResourceLimits, PhaseInstrumentation)
- Method execution (MethodExecutor)
- Orchestrator (the main 11-phase orchestration engine)

The Orchestrator is the sole owner of the provider; processors and executors
receive pre-prepared data.
"""

from __future__ import annotations

import asyncio
import hashlib
import inspect
import json
import logging
import os
import statistics
```

```
import threading
import time
from collections import deque
from collections.abc import Callable
from dataclasses import asdict, dataclass, field, is_dataclass
from datetime import datetime
from pathlib import Path
from types import MappingProxyType
from typing import TYPE_CHECKING, Any, Literal, ParamSpec, TypedDict, TypeVar

if TYPE_CHECKING:
    from collections.abc import Callable

    from .factory import CanonicalQuestionnaire

from ...analysis.recommendation_engine import RecommendationEngine
from ...config.paths import PROJECT_ROOT, RULES_DIR, CONFIG_DIR
from ...processing.aggregation import (
    AggregationSettings,
    AreaPolicyAggregator,
    AreaScore,
    ClusterAggregator,
    ClusterScore,
    DimensionAggregator,
    DimensionScore,
    MacroAggregator,
    MacroScore,
    ValidationError,
    group_by,
    validate_scored_results,
)
from ..dependency_lockdown import get_dependency_lockdown
from . import executors_contract as executors
from .arg_router import ArgRouterError, ArgumentValidationError, ExtendedArgRouter
from .class_registry import ClassRegistryError, build_class_registry
from .executor_config import ExecutorConfig
from .versions import CALIBRATION_VERSION
from ...utils.paths import safe_join

logger = logging.getLogger(__name__)
_CORE_MODULE_DIR = Path(__file__).resolve().parent


def resolve_workspace_path(
    path: str | Path,
    *,
    project_root: Path = PROJECT_ROOT,
    rules_dir: Path = RULES_DIR,
    module_dir: Path = _CORE_MODULE_DIR,
) -> Path:
    """Resolve repository-relative paths deterministically."""
    path_obj = Path(path)

    if path_obj.is_absolute():
        return path_obj

    sanitized = safe_join(project_root, *path_obj.parts)
    candidates = [
        sanitized,
        safe_join(module_dir, *path_obj.parts),
        safe_join(rules_dir, *path_obj.parts),
    ]

    if not path_obj.parts or path_obj.parts[0] != "rules":
        candidates.append(safe_join(rules_dir, "METODOS", *path_obj.parts))

    for candidate in candidates:
        if candidate.exists():
```

```
            return candidate

    return sanitized

# Environment-configurable expectations for validation
EXPECTED_QUESTION_COUNT = int(os.getenv("EXPECTED_QUESTION_COUNT", "305"))
EXPECTED_METHOD_COUNT = int(os.getenv("EXPECTED_METHOD_COUNT", "416"))
PHASE_TIMEOUT_DEFAULT = int(os.getenv("PHASE_TIMEOUT_SECONDS", "300"))
P01_EXPECTED_CHUNK_COUNT = 60


class PhaseTimeoutError(RuntimeError):
    """Raised when a phase exceeds its timeout."""

    def __init__(self, phase_id: int | str, phase_name: str, timeout_s: float) -> None:
        self.phase_id = phase_id
        self.phase_name = phase_name
        self.timeout_s = timeout_s
        super().__init__(
            f"Phase {phase_id} ({phase_name}) timed out after {timeout_s}s"
        )


# ParamSpec and TypeVar for execute_phase_with_timeout
P = ParamSpec("P")
T = TypeVar("T")


async def execute_phase_with_timeout(
    phase_id: int,
    phase_name: str,
    coro: Callable[P, T] | None = None,
    *varargs: P.args,
    handler: Callable[P, T] | None = None,  # Legacy parameter for backward compatibility
    args: tuple | None = None,  # Legacy parameter for backward compatibility
    timeout_s: float = 300.0,
    **kwargs: P.kwargs,
) -> T:
    """Execute an async phase with timeout and comprehensive logging.

    Args:
        phase_id: Numeric phase identifier
        phase_name: Human-readable phase name
        coro: Coroutine/callable to execute (preferred)
        *varargs: Positional arguments for coro (when using positional style)
        handler: Legacy alias for coro (for backward compatibility)
        args: Legacy parameter for positional arguments (for backward compatibility)
        timeout_s: Timeout in seconds (default: 300.0)
        **kwargs: Keyword arguments for coro

    Returns:
        Result from coro

    Raises:
        PhaseTimeoutError: If execution exceeds timeout_s
        Exception: Any exception raised by coro
        ValueError: If neither coro nor handler is provided
    """
    # Support both coro and handler (legacy) parameter names
    target = coro or handler
    if target is None:
        raise ValueError("Either 'coro' or 'handler' must be provided")

    # Support both varargs (*args in signature) and args kwarg (legacy)
    call_args = varargs if varargs else (args or ())

    start = time.perf_counter()
    logger.info(
```

```
        "phase_execution_started",
        extra={"phase_id": phase_id, "phase_name": phase_name, "timeout_s": timeout_s},
    )
    try:
        result = await asyncio.wait_for(target(*call_args, **kwargs), timeout=timeout_s)
        elapsed = time.perf_counter() - start
        logger.info(
            "phase_execution_completed",
            extra={
                "phase_id": phase_id,
                "phase_name": phase_name,
                "elapsed_s": elapsed,
                "timeout_s": timeout_s,
                "time_remaining_s": timeout_s - elapsed,
            },
        )
        return result
    except asyncio.TimeoutError as exc:
        elapsed = time.perf_counter() - start
        logger.error(
            "phase_execution_timeout",
            extra={
                "phase_id": phase_id,
                "phase_name": phase_name,
                "elapsed_s": elapsed,
                "timeout_s": timeout_s,
                "exceeded_by_s": elapsed - timeout_s,
            },
        )
        raise PhaseTimeoutError(phase_id, phase_name, timeout_s) from exc
    except asyncio.CancelledError:
        elapsed = time.perf_counter() - start
        logger.warning(
            "phase_execution_cancelled",
            extra={
                "phase_id": phase_id,
                "phase_name": phase_name,
                "elapsed_s": elapsed,
            },
        )
        raise  # Re-raise to propagate cancellation
    except Exception as exc:
        elapsed = time.perf_counter() - start
        logger.error(
            "phase_execution_error",
            extra={
                "phase_id": phase_id,
                "phase_name": phase_name,
                "elapsed_s": elapsed,
                "error_type": type(exc).__name__,
                "error_message": str(exc),
            },
            exc_info=True,
        )
        raise


def _normalize_monolith_for_hash(monolith: dict | MappingProxyType) -> dict:
    """Normalize monolith for hash computation and JSON serialization.

    Converts MappingProxyType to dict recursively to ensure:
    1. JSON serialization doesn't fail
    2. Hash computation is consistent

    Args:
        monolith: Monolith data (may be MappingProxyType or dict)

    Returns:
```

```
        Normalized dict suitable for hashing and JSON serialization

    Raises:
        RuntimeError: If normalization fails or produces inconsistent results
    """
    if isinstance(monolith, MappingProxyType):
        monolith = dict(monolith)

    # Deep-convert nested mapping proxies if they exist
    def _convert(obj: Any) -> Any:
        if isinstance(obj, MappingProxyType):
            obj = dict(obj)
        if isinstance(obj, dict):
            return {k: _convert(v) for k, v in obj.items()}
        if isinstance(obj, list):
            return [_convert(v) for v in obj]
        return obj

    normalized = _convert(monolith)

    # Verify normalization is idempotent
    try:
        # Test that we can serialize it
        json.dumps(normalized, sort_keys=True, ensure_ascii=False, separators=(",", ":"))
    except (TypeError, ValueError) as exc:
        raise RuntimeError(f"Monolith normalization failed: {exc}") from exc

    return normalized


class MacroScoreDict(TypedDict):
    """Typed container for macro score evaluation results."""
    macro_score: MacroScore
    macro_score_normalized: float
    cluster_scores: list[ClusterScore]
    cross_cutting_coherence: float
    systemic_gaps: list[str]
    strategic_alignment: float
    quality_band: str


@dataclass
class ClusterScoreData:
    """Type-safe cluster score data for macro evaluation."""
    id: str
    score: float
    normalized_score: float


@dataclass
class MacroEvaluation:
    """Type-safe macro evaluation result.

    This replaces polymorphic dict/object handling with a strict contract.
    All downstream consumers must treat macro scores as this type.
    """
    macro_score: float
    macro_score_normalized: float
    clusters: list[ClusterScoreData]


@dataclass(frozen=True)
class ChunkData:
    """Single semantic chunk from SPC (Smart Policy Chunks).

    Preserves chunk structure and metadata from the ingestion pipeline,
    enabling chunk-aware executor routing and scoped processing.
    """
```

```
    id: int
    text: str
    chunk_type: Literal["diagnostic", "activity", "indicator", "resource", "temporal",
"entity"]
    sentences: list[int]  # Global sentence IDs in this chunk
    tables: list[int]     # Global table IDs in this chunk
    start_pos: int
    end_pos: int
    confidence: float
    edges_out: list[int] = field(default_factory=list)  # Chunk IDs this connects to
    edges_in: list[int] = field(default_factory=list)   # Chunk IDs connecting to this


@dataclass
class PreprocessedDocument:
    """Orchestrator representation of a processed document.

    This is the normalized document format used internally by the orchestrator.
    It can be constructed from ingestion payloads or created directly.

    New in SPC exploitation: Preserves chunk structure when processing_mode='chunked',
    enabling chunk-aware executor routing and reducing redundant processing.
    """
    document_id: str
    raw_text: str
    sentences: list[Any]
    tables: list[Any]
    metadata: dict[str, Any]
    sentence_metadata: list[Any] = field(default_factory=list)
    indexes: dict[str, Any] | None = None
    structured_text: dict[str, Any] | None = None
    language: str | None = None
    ingested_at: datetime | None = None
    full_text: str | None = None

    # NEW CHUNK FIELDS for SPC exploitation
    chunks: list[ChunkData] = field(default_factory=list)
    chunk_index: dict[str, int] = field(default_factory=dict)  # Fast lookup: entity_id →
chunk_id
    chunk_graph: dict[str, Any] = field(default_factory=dict)  # Exposed graph structure
    processing_mode: Literal["flat", "chunked"] = "flat"  # Mode flag for backward
compatibility

    def __post_init__(self) -> None:
        """Validate document fields after initialization.

        Raises:
            ValueError: If raw_text is empty or whitespace-only
        """
        if (not self.raw_text or not self.raw_text.strip()) and self.full_text:
            # Backward-compatible fallback when only full_text is provided
            self.raw_text = self.full_text
        if not self.raw_text or not self.raw_text.strip():
            raise ValueError(
                "PreprocessedDocument cannot have empty raw_text. "
                "Use PreprocessedDocument.ensure() to create from SPC pipeline."
            )

    @staticmethod
    def _dataclass_to_dict(value: Any) -> Any:
        """Convert a dataclass to a dictionary if applicable."""
        if is_dataclass(value):
            return asdict(value)
        return value

    @classmethod
    def ensure(
        cls, document: Any, *, document_id: str | None = None, use_spc_ingestion: bool =
```

```
True
    ) -> PreprocessedDocument:
        """Normalize arbitrary ingestion payloads into orchestrator documents.

        Args:
            document: Document to normalize (PreprocessedDocument or CanonPolicyPackage)
            document_id: Optional document ID override
            use_spc_ingestion: Must be True (SPC is now the only supported ingestion
method)

        Returns:
            PreprocessedDocument instance

        Raises:
            ValueError: If use_spc_ingestion is False
            TypeError: If document type is not supported
        """
        # Enforce SPC-only ingestion
        if not use_spc_ingestion:
            raise ValueError(
                "SPC ingestion is now required. Set use_spc_ingestion=True or remove the
parameter. "
                "Legacy ingestion methods (document_ingestion module) are no longer
supported."
            )

        # Reject class types - only accept instances
        if isinstance(document, type):
            class_name = getattr(document, '__name__', str(document))
            raise TypeError(
                f"Expected document instance, got class type '{class_name}'. "
                "Pass an instance of the document, not the class itself."
            )

        if isinstance(document, cls):
            return document

        # Check for SPC (Smart Policy Chunks) ingestion - canonical phase-one
        # Documents must have chunk_graph attribute (from CanonPolicyPackage)
        if hasattr(document, "chunk_graph"):
            # Validate chunk_graph exists and is not empty
            chunk_graph = getattr(document, "chunk_graph", None)
            if chunk_graph is None:
                raise ValueError(
                    "Document has chunk_graph attribute but it is None. "
                    "Ensure SPC ingestion pipeline completed successfully."
                )

            # Validate chunk_graph has chunks
            if not hasattr(chunk_graph, 'chunks') or not chunk_graph.chunks:
                raise ValueError(
                    "Document chunk_graph is empty. "
                    "Ensure SPC ingestion pipeline completed successfully and extracted
chunks."
                )

            try:
                from saaaaaa.utils.spc_adapter import SPCAdapter
                adapter = SPCAdapter()
                preprocessed = adapter.to_preprocessed_document(document,
document_id=document_id)

                # Comprehensive SPC ingestion validation
                validation_results = []

                # Validate raw_text
                if not preprocessed.raw_text or not preprocessed.raw_text.strip():
                    raise ValueError(
```

```
                    "SPC ingestion produced empty document. "
                    "Check that the source document contains extractable text."
                )
            text_length = len(preprocessed.raw_text)
            validation_results.append(f"raw_text: {text_length} chars")

            # Validate sentences extracted
            sentence_count = len(preprocessed.sentences) if preprocessed.sentences
else 0
            if sentence_count == 0:
                logger.warning("SPC ingestion produced zero sentences - document may
be malformed")
            validation_results.append(f"sentences: {sentence_count}")

            # Validate chunk_graph exists
            chunk_count = preprocessed.metadata.get("chunk_count", 0)
            validation_results.append(f"chunks: {chunk_count}")

            # Log successful validation
            logger.info(f"SPC ingestion validation passed: {',
'.join(validation_results)}")

            return preprocessed
        except ImportError as e:
            raise ImportError(
                "SPC ingestion requires spc_adapter module. "
                "Ensure saaaaaa.utils.spc_adapter is available."
            ) from e
        except ValueError:
            # Re-raise ValueError directly (e.g., empty document validation)
            raise
        except Exception as e:
            raise TypeError(
                f"Failed to adapt SPC document: {e}. "
                "Ensure document is a valid CanonPolicyPackage instance from SPC
pipeline."
            ) from e

    raise TypeError(
        "Unsupported preprocessed document payload. "
        f"Expected PreprocessedDocument or CanonPolicyPackage with chunk_graph, got
{type(document)!r}. "
        "Documents must be processed through the SPC ingestion pipeline first."
    )

@dataclass
class Evidence:
    """Evidence container for orchestrator results."""
    modality: str
    elements: list[Any] = field(default_factory=list)
    raw_results: dict[str, Any] = field(default_factory=dict)

class AbortRequested(RuntimeError):
    """Raised when an abort signal is triggered during orchestration."""

class AbortSignal:
    """Thread-safe abort signal shared across orchestration phases."""

    def __init__(self) -> None:
        self._event = threading.Event()
        self._lock = threading.Lock()
        self._reason: str | None = None
        self._timestamp: datetime | None = None

    def abort(self, reason: str) -> None:
        """Trigger an abort with a reason and timestamp."""
        if not reason:
            reason = "Abort requested"
```

```
        with self._lock:
            if not self._event.is_set():
                self._event.set()
                self._reason = reason
                self._timestamp = datetime.utcnow()

    def is_aborted(self) -> bool:
        """Check whether abort has been triggered."""
        return self._event.is_set()

    def get_reason(self) -> str | None:
        """Return the abort reason if set."""
        with self._lock:
            return self._reason

    def get_timestamp(self) -> datetime | None:
        """Return the abort timestamp if set."""
        with self._lock:
            return self._timestamp

    def reset(self) -> None:
        """Clear the abort signal."""
        with self._lock:
            self._event.clear()
            self._reason = None
            self._timestamp = None

class ResourceLimits:
    """Runtime resource guard with adaptive worker prediction."""

    def __init__(
        self,
        max_memory_mb: float | None = 4096.0,
        max_cpu_percent: float = 85.0,
        max_workers: int = 32,
        min_workers: int = 4,
        hard_max_workers: int = 64,
        history: int = 120,
    ) -> None:
        self.max_memory_mb = max_memory_mb
        self.max_cpu_percent = max_cpu_percent
        self.min_workers = max(1, min_workers)
        self.hard_max_workers = max(self.min_workers, hard_max_workers)
        self._max_workers = max(self.min_workers, min(max_workers, self.hard_max_workers))
        self._usage_history: deque[dict[str, float]] = deque(maxlen=history)
        self._semaphore: asyncio.Semaphore | None = None
        self._semaphore_limit = self._max_workers
        self._async_lock: asyncio.Lock | None = None
        self._psutil = None
        self._psutil_process = None
        try:  # pragma: no cover - optional dependency
            import psutil  # type: ignore[import-untyped]

            self._psutil = psutil
            self._psutil_process = psutil.Process(os.getpid())
        except Exception:  # pragma: no cover - psutil missing
            self._psutil = None
            self._psutil_process = None

    @property
    def max_workers(self) -> int:
        """Return the current worker budget."""
        return self._max_workers

    def attach_semaphore(self, semaphore: asyncio.Semaphore) -> None:
        """Attach an asyncio semaphore for budget control."""
        self._semaphore = semaphore
        self._semaphore_limit = self._max_workers
```

```
    async def apply_worker_budget(self) -> int:
        """Apply the current worker budget to the semaphore."""
        if self._semaphore is None:
            return self._max_workers

        if self._async_lock is None:
            self._async_lock = asyncio.Lock()

        async with self._async_lock:
            desired = self._max_workers
            current = self._semaphore_limit
            if desired > current:
                for _ in range(desired - current):
                    self._semaphore.release()
            elif desired < current:
                reduction = current - desired
                for _ in range(reduction):
                    await self._semaphore.acquire()
            self._semaphore_limit = desired
            return self._max_workers

    def _record_usage(self, usage: dict[str, float]) -> None:
        """Record resource usage and predict worker budget."""
        self._usage_history.append(usage)
        self._predict_worker_budget()

    def _predict_worker_budget(self) -> None:
        """Adjust worker budget based on recent resource usage."""
        if len(self._usage_history) < 5:
            return

        cpu_vals = [entry["cpu_percent"] for entry in self._usage_history]
        mem_vals = [entry["memory_percent"] for entry in self._usage_history]
        recent_cpu = cpu_vals[-5:]
        recent_mem = mem_vals[-5:]
        avg_cpu = statistics.mean(recent_cpu)
        avg_mem = statistics.mean(recent_mem)

        new_budget = self._max_workers
        if self.max_cpu_percent and avg_cpu > self.max_cpu_percent * 0.95 or
self.max_memory_mb and avg_mem > 90.0:
            new_budget = max(self.min_workers, self._max_workers - 1)
        elif avg_cpu < self.max_cpu_percent * 0.6 and avg_mem < 70.0:
            new_budget = min(self.hard_max_workers, self._max_workers + 1)

        self._max_workers = max(self.min_workers, min(new_budget, self.hard_max_workers))

    def check_memory_exceeded(
        self, usage: dict[str, float] | None = None
    ) -> tuple[bool, dict[str, float]]:
        """Check if memory limit has been exceeded."""
        usage = usage or self.get_resource_usage()
        exceeded = False
        if self.max_memory_mb is not None:
            exceeded = usage.get("rss_mb", 0.0) > self.max_memory_mb
        return exceeded, usage

    def check_cpu_exceeded(
        self, usage: dict[str, float] | None = None
    ) -> tuple[bool, dict[str, float]]:
        """Check if CPU limit has been exceeded."""
        usage = usage or self.get_resource_usage()
        exceeded = False
        if self.max_cpu_percent:
            exceeded = usage.get("cpu_percent", 0.0) > self.max_cpu_percent
        return exceeded, usage
```

```
    def get_resource_usage(self) -> dict[str, float]:
        """Capture current resource usage metrics."""
        timestamp = datetime.utcnow().isoformat()
        cpu_percent = 0.0
        memory_percent = 0.0
        rss_mb = 0.0

        if self._psutil:
            try:  # pragma: no cover - psutil branch
                cpu_percent = float(self._psutil.cpu_percent(interval=None))
                virtual_memory = self._psutil.virtual_memory()
                memory_percent = float(virtual_memory.percent)
                if self._psutil_process is not None:
                    rss_mb = float(self._psutil_process.memory_info().rss / (1024 * 1024))
            except Exception:
                cpu_percent = 0.0
        else:
            try:
                load1, _, _ = os.getloadavg()
                cpu_percent = float(min(100.0, load1 * 100))
            except OSError:
                cpu_percent = 0.0
            try:
                import resource

                usage_info = resource.getrusage(resource.RUSAGE_SELF)
                rss_mb = float(usage_info.ru_maxrss / 1024)
            except Exception:
                rss_mb = 0.0

        usage = {
            "timestamp": timestamp,
            "cpu_percent": cpu_percent,
            "memory_percent": memory_percent,
            "rss_mb": rss_mb,
            "worker_budget": float(self._max_workers),
        }
        self._record_usage(usage)
        return usage

    def get_usage_history(self) -> list[dict[str, float]]:
        """Return the recorded usage history."""
        return list(self._usage_history)

class PhaseInstrumentation:
    """Collects granular telemetry for each orchestration phase."""

    def __init__(
        self,
        phase_id: int,
        name: str,
        items_total: int | None = None,
        snapshot_interval: int = 10,
        resource_limits: ResourceLimits | None = None,
    ) -> None:
        self.phase_id = phase_id
        self.name = name
        self.items_total = items_total or 0
        self.snapshot_interval = max(1, snapshot_interval)
        self.resource_limits = resource_limits
        self.items_processed = 0
        self.start_time: float | None = None
        self.end_time: float | None = None
        self.warnings: list[dict[str, Any]] = []
        self.errors: list[dict[str, Any]] = []
        self.resource_snapshots: list[dict[str, Any]] = []
        self.latencies: list[float] = []
        self.anomalies: list[dict[str, Any]] = []
```

```
def start(self, items_total: int | None = None) -> None:
    """Mark the start of phase execution."""
    if items_total is not None:
        self.items_total = items_total
    self.start_time = time.perf_counter()

def increment(self, count: int = 1, latency: float | None = None) -> None:
    """Increment processed item count and optionally record latency."""
    self.items_processed += count
    if latency is not None:
        self.latencies.append(latency)
        self._detect_latency_anomaly(latency)
    if self.resource_limits and self.should_snapshot():
        self.capture_resource_snapshot()

def should_snapshot(self) -> bool:
    """Determine if a resource snapshot should be captured."""
    if self.items_total == 0:
        return False
    if self.items_processed == 0:
        return False
    return self.items_processed % self.snapshot_interval == 0

def capture_resource_snapshot(self) -> None:
    """Capture a resource usage snapshot."""
    if not self.resource_limits:
        return
    snapshot = self.resource_limits.get_resource_usage()
    snapshot["items_processed"] = self.items_processed
    self.resource_snapshots.append(snapshot)

def record_warning(self, category: str, message: str, **extra: Any) -> None:
    """Record a warning during phase execution."""
    entry = {
        "category": category,
        "message": message,
        **extra,
        "timestamp": datetime.utcnow().isoformat(),
    }
    self.warnings.append(entry)

def record_error(self, category: str, message: str, **extra: Any) -> None:
    """Record an error during phase execution."""
    entry = {
        "category": category,
        "message": message,
        **extra,
        "timestamp": datetime.utcnow().isoformat(),
    }
    self.errors.append(entry)

def _detect_latency_anomaly(self, latency: float) -> None:
    """Detect latency anomalies using statistical thresholds."""
    if len(self.latencies) < 5:
        return
    mean_latency = statistics.mean(self.latencies)
    std_latency = statistics.pstdev(self.latencies) or 0.0
    threshold = mean_latency + (3 * std_latency)
    if std_latency and latency > threshold:
        self.anomalies.append(
            {
                "type": "latency_spike",
                "latency": latency,
                "mean": mean_latency,
                "std": std_latency,
                "timestamp": datetime.utcnow().isoformat(),
            }
```

```
        )

    def complete(self) -> None:
        """Mark the end of phase execution."""
        self.end_time = time.perf_counter()

    def duration_ms(self) -> float | None:
        """Return the phase duration in milliseconds."""
        if self.start_time is None or self.end_time is None:
            return None
        return (self.end_time - self.start_time) * 1000.0

    def progress(self) -> float | None:
        """Return the progress fraction (0.0 to 1.0)."""
        if not self.items_total:
            return None
        return min(1.0, self.items_processed / float(self.items_total))

    def throughput(self) -> float | None:
        """Return items processed per second."""
        if self.start_time is None:
            return None
        elapsed = (
            (time.perf_counter() - self.start_time)
            if self.end_time is None
            else (self.end_time - self.start_time)
        )
        if not elapsed:
            return None
        return self.items_processed / elapsed

    def latency_histogram(self) -> dict[str, float | None]:
        """Return latency percentiles."""
        if not self.latencies:
            return {"p50": None, "p95": None, "p99": None}
        sorted_latencies = sorted(self.latencies)

        def percentile(p: float) -> float:
            if not sorted_latencies:
                return 0.0
            k = (len(sorted_latencies) - 1) * (p / 100.0)
            f = int(k)
            c = min(f + 1, len(sorted_latencies) - 1)
            if f == c:
                return sorted_latencies[int(k)]
            d0 = sorted_latencies[f] * (c - k)
            d1 = sorted_latencies[c] * (k - f)
            return d0 + d1

        return {
            "p50": percentile(50.0),
            "p95": percentile(95.0),
            "p99": percentile(99.0),
        }

    def build_metrics(self) -> dict[str, Any]:
        """Build a metrics summary dictionary."""
        return {
            "phase_id": self.phase_id,
            "name": self.name,
            "duration_ms": self.duration_ms(),
            "items_processed": self.items_processed,
            "items_total": self.items_total,
            "progress": self.progress(),
            "throughput": self.throughput(),
            "warnings": list(self.warnings),
            "errors": list(self.errors),
            "resource_snapshots": list(self.resource_snapshots),
```

```
            "latency_histogram": self.latency_histogram(),
            "anomalies": list(self.anomalies),
        }

@dataclass
class PhaseResult:
    """Result of a single orchestration phase."""
    success: bool
    phase_id: str
    data: Any
    error: Exception | None
    duration_ms: float
    mode: str
    aborted: bool = False

@dataclass
class MicroQuestionRun:
    """Result of executing a single micro-question."""
    question_id: str
    question_global: int
    base_slot: str
    metadata: dict[str, Any]
    evidence: Evidence | None
    error: str | None = None
    duration_ms: float | None = None
    aborted: bool = False

@dataclass
class ScoredMicroQuestion:
    """Scored micro-question result."""
    question_id: str
    question_global: int
    base_slot: str
    score: float | None
    normalized_score: float | None
    quality_level: str | None
    evidence: Evidence | None
    scoring_details: dict[str, Any] = field(default_factory=dict)
    metadata: dict[str, Any] = field(default_factory=dict)
    error: str | None = None


class _LazyInstanceDict:
    """Lazy instance dictionary for backward compatibility.

    Provides dict-like interface but delegates to MethodRegistry
    for lazy instantiation. This maintains compatibility with code
    that accesses MethodExecutor.instances directly.
    """

    def __init__(self, method_registry: Any) -> None:
        self._registry = method_registry

    def get(self, class_name: str, default: Any = None) -> Any:
        """Get instance lazily."""
        try:
            return self._registry._get_instance(class_name)
        except Exception:
            return default

    def __getitem__(self, class_name: str) -> Any:
        """Get instance lazily (dict access)."""
        return self._registry._get_instance(class_name)

    def __contains__(self, class_name: str) -> bool:
        """Check if class is available."""
        return class_name in self._registry._class_paths
```

```
    def keys(self) -> list[str]:
        """Get available class names."""
        return list(self._registry._class_paths.keys())

    def values(self) -> list[Any]:
        """Get instantiated instances (triggers lazy loading)."""
        return [self.get(name) for name in self.keys()]

    def items(self) -> list[tuple[str, Any]]:
        """Get (name, instance) pairs (triggers lazy loading)."""
        return [(name, self.get(name)) for name in self.keys()]

    def __len__(self) -> int:
        """Get number of available classes."""
        return len(self._registry._class_paths)


class MethodExecutor:
    """Execute catalog methods using lazy method injection.

    This executor uses MethodRegistry for lazy instantiation:
    - Classes are loaded only when their methods are first called
    - Failed classes don't block other methods from working
    - Methods can be directly injected without classes
    - Instance caching for efficiency

    No upfront class instantiation - lightweight and decoupled.
    """

    def __init__(
        self,
        dispatcher: Any | None = None, # dispatcher is deprecated
        signal_registry: Any | None = None,
        method_registry: Any | None = None, # MethodRegistry instance
    ) -> None:
        from .method_registry import MethodRegistry, setup_default_instantiation_rules

        self.degraded_mode = False
        self.degraded_reasons: list[str] = []
        self.signal_registry = signal_registry

        # Initialize method registry with lazy loading
        if method_registry is not None:
            self._method_registry = method_registry
        else:
            try:
                self._method_registry = MethodRegistry()
                setup_default_instantiation_rules(self._method_registry)
                logger.info("method_registry_initialized_lazy_mode")
            except Exception as exc:
                self.degraded_mode = True
                reason = f"Method registry initialization failed: {exc}"
                self.degraded_reasons.append(reason)
                logger.error("DEGRADED MODE: %s", reason)
                # Create empty registry for graceful degradation
                self._method_registry = MethodRegistry(class_paths={})

        # Build minimal class type registry for ArgRouter compatibility
        # Note: This doesn't instantiate classes, just loads types
        try:
            from .class_registry import build_class_registry
            registry = build_class_registry()
        except (ClassRegistryError, ModuleNotFoundError, ImportError) as exc:
            self.degraded_mode = True
            reason = f"Could not build class registry: {exc}"
            self.degraded_reasons.append(reason)
            logger.warning("DEGRADED MODE: %s", reason)
            registry = {}
```

```
        # Create ExtendedArgRouter with the registry for enhanced validation and metrics
        self._router = ExtendedArgRouter(registry)
        self.instances = _LazyInstanceDict(self._method_registry)

    @staticmethod
    def _supports_parameter(callable_obj: Any, parameter_name: str) -> bool:
        try:
            signature = inspect.signature(callable_obj)
        except (TypeError, ValueError):  # pragma: no cover - builtins / C extensions
            return False
        return parameter_name in signature.parameters

    def execute(self, class_name: str, method_name: str, **kwargs: Any) -> Any:
        """Execute a method using lazy instantiation.

        Args:
            class_name: Name of the class.
            method_name: Name of the method to execute.
            **kwargs: Keyword arguments to pass to the method call.

        Returns:
            The method's return value.

        Raises:
            ArgRouterError: If routing fails
            AttributeError: If method doesn't exist
            MethodRegistryError: If method cannot be retrieved
        """
        from .method_registry import MethodRegistryError

        # Get method from registry (lazy instantiation)
        try:
            method = self._method_registry.get_method(class_name, method_name)
        except MethodRegistryError as exc:
            logger.error(
                "method_retrieval_failed",
                class_name=class_name,
                method_name=method_name,
                error=str(exc),
            )
            # Graceful degradation - return None for missing methods
            if self.degraded_mode:
                logger.warning("Returning None due to degraded mode")
                return None
            raise AttributeError(
                f"Cannot retrieve {class_name}.{method_name}: {exc}"
            ) from exc

        # Route arguments and execute
        try:
            args, routed_kwargs = self._router.route(class_name, method_name,
dict(kwargs))
            return method(*args, **routed_kwargs)
        except (ArgRouterError, ArgumentValidationError):
            logger.exception("Argument routing failed for %s.%s", class_name, method_name)
            raise
        except Exception:
            logger.exception("Method execution failed for %s.%s", class_name, method_name)
            raise

    def inject_method(
        self,
        class_name: str,
        method_name: str,
        method: Callable[..., Any],
    ) -> None:
        """Inject a method directly without requiring a class.
```

```
        This allows you to provide custom implementations that bypass
        class instantiation entirely. Useful for:
        - Custom implementations
        - Mocking/testing
        - Hotfixes without modifying classes

        Example:
            def custom_analyzer(text: str, **kwargs) -> dict:
                return {"result": "custom analysis"}

            executor.inject_method("CustomClass", "analyze", custom_analyzer)

        Args:
            class_name: Virtual class name for routing
            method_name: Method name
            method: Callable to inject
        """
        self._method_registry.inject_method(class_name, method_name, method)
        logger.info(
            "method_injected_into_executor",
            class_name=class_name,
            method_name=method_name,
        )

    def has_method(self, class_name: str, method_name: str) -> bool:
        """Check if a method is available.

        Args:
            class_name: Class name
            method_name: Method name

        Returns:
            True if method exists or is injected
        """
        return self._method_registry.has_method(class_name, method_name)

    def get_registry_stats(self) -> dict[str, Any]:
        """Get statistics from the method registry.

        Returns:
            Dict with registry statistics including:
            - total_classes_registered: Total classes in registry
            - instantiated_classes: Classes that have been instantiated
            - failed_classes: Classes that failed instantiation
            - direct_methods_injected: Methods injected directly
        """
        return self._method_registry.get_stats()

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

        Returns:
            Dict with routing statistics including:
            - total_routes: Total number of routes processed
            - special_routes_hit: Count of special route invocations
            - validation_errors: Count of validation failures
            - silent_drops_prevented: Count of silent parameter drops prevented
        """
        if hasattr(self._router, 'get_metrics'):
            return self._router.get_metrics()
        return {}

def validate_phase_definitions(phase_list: list[tuple[int, str, str, str]],
orchestrator_class: type) -> None:
    """Validate phase definitions for structural coherence.

    This is a hard gate: if phase definitions are broken, the orchestrator cannot start.
```

```
    No "limited mode" is allowed when the base schema is corrupted.

    Args:
        phase_list: List of phase tuples (id, mode, handler, label)
        orchestrator_class: Orchestrator class to check for handler methods

    Raises:
        RuntimeError: If phase definitions are invalid
    """
    if not phase_list:
        raise RuntimeError("FASES cannot be empty - no phases defined for orchestration")

    # Extract phase IDs
    phase_ids = [phase[0] for phase in phase_list]

    # Check for duplicate phase IDs
    seen_ids = set()
    for phase_id in phase_ids:
        if phase_id in seen_ids:
            raise RuntimeError(
                f"Duplicate phase ID {phase_id} in FASES definition. "
                "Phase IDs must be unique."
            )
        seen_ids.add(phase_id)

    # Check that IDs are contiguous starting from 0
    # For performance: check sorted and validate range
    if phase_ids != sorted(phase_ids):
        raise RuntimeError(
            f"Phase IDs must be sorted in ascending order. Got {phase_ids}"
        )
    if phase_ids[0] != 0:
        raise RuntimeError(
            f"Phase IDs must start from 0. Got first ID: {phase_ids[0]}"
        )
    if phase_ids[-1] != len(phase_list) - 1:
        raise RuntimeError(
            f"Phase IDs must be contiguous from 0 to {len(phase_list) - 1}. "
            f"Got highest ID: {phase_ids[-1]}"
        )

    # Validate each phase
    valid_modes = {"sync", "async"}
    for phase_id, mode, handler_name, label in phase_list:
        # Validate mode
        if mode not in valid_modes:
            raise RuntimeError(
                f"Phase {phase_id} ({label}): invalid mode '{mode}'. "
                f"Mode must be one of {valid_modes}"
            )

        # Validate handler exists as method in orchestrator
        if not hasattr(orchestrator_class, handler_name):
            raise RuntimeError(
                f"Phase {phase_id} ({label}): handler method '{handler_name}' "
                f"does not exist in {orchestrator_class.__name__}"
            )

        # Validate handler is callable
        handler = getattr(orchestrator_class, handler_name, None)
        if not callable(handler):
            raise RuntimeError(
                f"Phase {phase_id} ({label}): handler '{handler_name}' "
                f"is not callable"
            )


class Orchestrator:
```

```
"""Robust 11-phase orchestrator with abort support and resource control.

The Orchestrator owns the provider and prepares all data for processors
and executors. It executes 11 phases synchronously or asynchronously,
with full instrumentation and abort capability.
"""

FASES: list[tuple[int, str, str, str]] = [
    (0, "sync", "_load_configuration", "FASE 0 - Validación de Configuración"),
    (1, "sync", "_ingest_document", "FASE 1 - Ingestión de Documento"),
    (2, "async", "_execute_micro_questions_async", "FASE 2 - Micro Preguntas"),
    (3, "async", "_score_micro_results_async", "FASE 3 - Scoring Micro"),
    (4, "async", "_aggregate_dimensions_async", "FASE 4 - Agregación Dimensiones"),
    (5, "async", "_aggregate_policy_areas_async", "FASE 5 - Agregación Áreas"),
    (6, "sync", "_aggregate_clusters", "FASE 6 - Agregación Clústeres"),
    (7, "sync", "_evaluate_macro", "FASE 7 - Evaluación Macro"),
    (8, "async", "_generate_recommendations", "FASE 8 - Recomendaciones"),
    (9, "sync", "_assemble_report", "FASE 9 - Ensamblado de Reporte"),
    (10, "async", "_format_and_export", "FASE 10 - Formateo y Exportación"),
]

PHASE_ITEM_TARGETS: dict[int, int] = {
    0: 1,
    1: 1,
    2: 300,
    3: 300,
    4: 60,
    5: 10,
    6: 4,
    7: 1,
    8: 1,
    9: 1,
    10: 1,
}

PHASE_OUTPUT_KEYS: dict[int, str] = {
    0: "config",
    1: "document",
    2: "micro_results",
    3: "scored_results",
    4: "dimension_scores",
    5: "policy_area_scores",
    6: "cluster_scores",
    7: "macro_result",
    8: "recommendations",
    9: "report",
    10: "export_payload",
}

PHASE_ARGUMENT_KEYS: dict[int, list[str]] = {
    1: ["pdf_path", "config"],
    2: ["document", "config"],
    3: ["micro_results", "config"],
    4: ["scored_results", "config"],
    5: ["dimension_scores", "config"],
    6: ["policy_area_scores", "config"],
    7: ["cluster_scores", "config"],
    8: ["macro_result", "config"],
    9: ["recommendations", "config"],
    10: ["report", "config"],
}

# Phase timeout configuration (in seconds)
PHASE_TIMEOUTS: dict[int, float] = {
    0: 60,     # Configuration validation
    1: 120,    # Document ingestion
    2: 600,    # Micro questions (300 items)
    3: 300,    # Scoring micro
```

```
    4: 180,    # Dimension aggregation
    5: 120,    # Policy area aggregation
    6: 60,     # Cluster aggregation
    7: 60,     # Macro evaluation
    8: 120,    # Recommendations
    9: 60,     # Report assembly
    10: 120,   # Format and export
}

# Score normalization constant
PERCENTAGE_SCALE: int = 100

def __init__(
    self,
    method_executor: MethodExecutor,
    questionnaire: CanonicalQuestionnaire,
    executor_config: "ExecutorConfig",
    calibration_orchestrator: Optional["CalibrationOrchestrator"] = None,
    resource_limits: ResourceLimits | None = None,
    resource_snapshot_interval: int = 10,
) -> None:
    """Initialize the orchestrator with all dependencies injected.

    Args:
        method_executor: A configured MethodExecutor instance.
        questionnaire: A loaded and validated CanonicalQuestionnaire instance.
        executor_config: The executor configuration object.
        calibration_orchestrator: The calibration orchestrator instance.
        resource_limits: Resource limit configuration.
        resource_snapshot_interval: Interval for resource snapshots.
    """
    from .factory import _validate_questionnaire_structure

    validate_phase_definitions(self.FASES, self.__class__)

    self.executor = method_executor
    self._canonical_questionnaire = questionnaire
    self._monolith_data = dict(questionnaire.data)
    self.executor_config = executor_config
    self.calibration_orchestrator = calibration_orchestrator
    self.resource_limits = resource_limits or ResourceLimits()
    self.resource_snapshot_interval = max(1, resource_snapshot_interval)
    from .factory import get_questionnaire_provider
    self.questionnaire_provider = get_questionnaire_provider()

    # Validate questionnaire structure
    try:
        _validate_questionnaire_structure(self._monolith_data)
    except (ValueError, TypeError) as e:
        raise RuntimeError(
            f"Questionnaire structure validation failed: {e}. "
            "Cannot start orchestrator with corrupt questionnaire."
        ) from e

    if not self.executor.instances:
        raise RuntimeError(
            "MethodExecutor.instances is empty - no executable methods registered."
        )

    self.executors = {
        "D1-Q1": executors.D1Q1_Executor, "D1-Q2": executors.D1Q2_Executor,
        "D1-Q3": executors.D1Q3_Executor, "D1-Q4": executors.D1Q4_Executor,
        "D1-Q5": executors.D1Q5_Executor, "D2-Q1": executors.D2Q1_Executor,
        "D2-Q2": executors.D2Q2_Executor, "D2-Q3": executors.D2Q3_Executor,
        "D2-Q4": executors.D2Q4_Executor, "D2-Q5": executors.D2Q5_Executor,
        "D3-Q1": executors.D3Q1_Executor, "D3-Q2": executors.D3Q2_Executor,
        "D3-Q3": executors.D3Q3_Executor, "D3-Q4": executors.D3Q4_Executor,
        "D3-Q5": executors.D3Q5_Executor, "D4-Q1": executors.D4Q1_Executor,
```

```
            "D4-Q2": executors.D4Q2_Executor, "D4-Q3": executors.D4Q3_Executor,
            "D4-Q4": executors.D4Q4_Executor, "D4-Q5": executors.D4Q5_Executor,
            "D5-Q1": executors.D5Q1_Executor, "D5-Q2": executors.D5Q2_Executor,
            "D5-Q3": executors.D5Q3_Executor, "D5-Q4": executors.D5Q4_Executor,
            "D5-Q5": executors.D5Q5_Executor, "D6-Q1": executors.D6Q1_Executor,
            "D6-Q2": executors.D6Q2_Executor, "D6-Q3": executors.D6Q3_Executor,
            "D6-Q4": executors.D6Q4_Executor, "D6-Q5": executors.D6Q5_Executor,
        }

        self.abort_signal = AbortSignal()
        self.phase_results: list[PhaseResult] = []
        self._phase_instrumentation: dict[int, PhaseInstrumentation] = {}
        self._phase_status: dict[int, str] = {
            phase_id: "not_started" for phase_id, *_ in self.FASES
        }
        self._phase_outputs: dict[int, Any] = {}
        self._context: dict[str, Any] = {}
        self._start_time: float | None = None

        self.dependency_lockdown = get_dependency_lockdown()
        logger.info(
            f"Orchestrator dependency mode:
{self.dependency_lockdown.get_mode_description()}"
        )

        try:
            self.recommendation_engine = RecommendationEngine(
                rules_path=RULES_DIR / "recommendation_rules_enhanced.json",
                schema_path=RULES_DIR / "recommendation_rules_enhanced.schema.json",
                questionnaire_provider=self.questionnaire_provider,
                orchestrator=self
            )
            logger.info("RecommendationEngine initialized with enhanced v2.0 rules")
        except Exception as e:
            logger.warning(f"Failed to initialize RecommendationEngine: {e}")
            self.recommendation_engine = None

    async def run(
        self,
        preprocessed_doc: Any,
        output_path: str | None = None,
        phase_timeout: float = 300,
        enable_cache: bool = True,
        progress_callback: Callable[[int, str, float], None] | None = None,
    ) -> dict[str, Any]:
        """Execute complete 11-phase orchestration pipeline with observability.

        This is the main entry point for orchestration, implementing:
        1. Real phase-by-phase execution (not simulated)
        2. OpenTelemetry spans for each phase
        3. Progress callbacks for UI/dashboard updates
        4. WiringValidator contract checks at boundaries
        5. Manifest generation for audit trail

        Args:
            preprocessed_doc: PreprocessedDocument from SPCAdapter
            output_path: Optional path to write final report
            phase_timeout: Timeout per phase in seconds
            enable_cache: Enable caching for expensive operations
            progress_callback: Optional callback(phase_num, phase_name, progress) for
real-time updates

        Returns:
            Dict with complete orchestration results:
                - macro_analysis: Macro-level scores
                - meso_analysis: Cluster-level scores
                - micro_analysis: Question-level scores
                - recommendations: Generated recommendations
```

```
            - report: Final assembled report
            - metadata: Pipeline metadata

        Raises:
            ValueError: If preprocessed_doc is invalid
            RuntimeError: If orchestration fails
        """
        from saaaaaa.observability import get_tracer, SpanKind

        tracer = get_tracer(__name__)

        # Start root span for entire orchestration
        with tracer.start_span("orchestration.run", kind=SpanKind.SERVER) as root_span:
            root_span.set_attribute("document_id", str(preprocessed_doc.document_id))
            root_span.set_attribute("phase_count", 11)
            root_span.set_attribute("cache_enabled", enable_cache)

            logger.info(
                "orchestration_started",
                document_id=preprocessed_doc.document_id,
                phase_count=11,
            )

            # Initialize result accumulator
            results = {
                "document_id": preprocessed_doc.document_id,
                "phases_completed": 0,
                "macro_analysis": None,
                "meso_analysis": None,
                "micro_analysis": None,
                "recommendations": None,
                "report": None,
                "metadata": {
                    "orchestrator_version": "2.0",
                    "start_time": datetime.now().isoformat(),
                },
            }

            try:
                # Phase 0: Configuration validation (already done in __init__)
                if progress_callback:
                    progress_callback(0, "Configuration Validation", 0.0)

                with tracer.start_span("phase.0.configuration", kind=SpanKind.INTERNAL) as
 span:
                    span.set_attribute("phase_id", 0)
                    span.set_attribute("phase_name", "Configuration Validation")
                    logger.info("phase_start", phase=0, name="Configuration Validation")

                    # Validate catalog is loaded
                    if self.catalog is None:
                        raise RuntimeError(
                            "Catalog not loaded. Cannot execute orchestration without
method catalog."
                        )

                    logger.info("phase_complete", phase=0)
                    results["phases_completed"] = 1

                # Phase 1: Document ingestion (already complete - validate adapter
contract)
                if progress_callback:
                    progress_callback(1, "Document Ingestion Validation", 9.1)

                with tracer.start_span("phase.1.ingestion_validation",
kind=SpanKind.INTERNAL) as span:
                    span.set_attribute("phase_id", 1)
                    span.set_attribute("phase_name", "Document Ingestion Validation")
```

```
                span.set_attribute("sentence_count", len(preprocessed_doc.sentences))

                logger.info("phase_start", phase=1, name="Document Ingestion
Validation")

                # Runtime validation: Adapter → Orchestrator contract
                try:
                    from saaaaaa.core.wiring.validation import WiringValidator
                    validator = WiringValidator()

                    preprocessed_dict = {
                        "document_id": preprocessed_doc.document_id,
                        "full_text": preprocessed_doc.full_text,
                        "sentences": list(preprocessed_doc.sentences),
                        "language": preprocessed_doc.language,
                        "sentence_count": len(preprocessed_doc.sentences),
                        "has_structured_text": preprocessed_doc.structured_text is not
 None,
                        "has_indexes": preprocessed_doc.indexes is not None,
                    }

                    validator.validate_adapter_to_orchestrator(preprocessed_dict)
                    logger.info("✓ Adapter → Orchestrator contract validated")
                except ImportError:
                    logger.warning("WiringValidator not available, skipping contract
validation")
                except Exception as e:
                    logger.error(f"Contract validation failed: {e}")
                    raise RuntimeError(f"Adapter → Orchestrator contract violation:
{e}") from e

                logger.info("phase_complete", phase=1)
                results["phases_completed"] = 2

            # Phase 2-10: Execute remaining phases
            # NOTE: Full phase implementation would call handler methods from FASES
            # For now, we'll create placeholder structure that real methods can
populate

            phase_definitions = [
                (2, "Micro Questions", "micro_analysis", 18.2),
                (3, "Scoring Micro", "scored_micro", 27.3),
                (4, "Dimension Aggregation", "dimension_scores", 36.4),
                (5, "Policy Area Aggregation", "policy_area_scores", 45.5),
                (6, "Cluster Aggregation", "cluster_scores", 54.5),
                (7, "Macro Evaluation", "macro_analysis", 63.6),
                (8, "Recommendations", "recommendations", 72.7),
                (9, "Report Assembly", "report", 81.8),
                (10, "Export", "export_payload", 90.9),
            ]

            for phase_id, phase_name, output_key, progress in phase_definitions:
                if progress_callback:
                    progress_callback(phase_id, phase_name, progress)

                with tracer.start_span(f"phase.{phase_id}.{output_key}",
kind=SpanKind.INTERNAL) as span:
                    span.set_attribute("phase_id", phase_id)
                    span.set_attribute("phase_name", phase_name)

                    logger.info("phase_start", phase=phase_id, name=phase_name)

                    # Execute phase handler if it exists
                    phase_tuple = next((p for p in self.FASES if p[0] == phase_id),
None)
                    if phase_tuple:
                        _, mode, handler_name, _ = phase_tuple
```

```
                    # Check if handler exists
                    if hasattr(self, handler_name):
                        handler = getattr(self, handler_name)

                        # Execute handler based on mode
                        try:
                            if mode == "async":
                                # Async handler - await it
                                phase_output = await
handler(preprocessed_doc=preprocessed_doc)
                            else:
                                # Sync handler
                                phase_output =
handler(preprocessed_doc=preprocessed_doc)

                            # Store output
                            self._phase_outputs[phase_id] = phase_output
                            results[output_key] = phase_output

                            span.set_attribute("phase_success", True)
                            logger.info("phase_complete", phase=phase_id,
output_size=len(str(phase_output)))
                        except Exception as e:
                            logger.error(f"Phase {phase_id} handler failed: {e}")
                            span.set_attribute("phase_success", False)
                            span.set_attribute("phase_error", str(e))
                            # Continue with empty output for now
                            results[output_key] = {"error": str(e), "phase":
phase_id}
                    else:
                        logger.warning(f"Phase {phase_id} handler '{handler_name}'
 not found")
                        results[output_key] = {"placeholder": True, "phase":
phase_id}
                else:
                    logger.warning(f"Phase {phase_id} not defined in FASES")
                    results[output_key] = {"placeholder": True, "phase": phase_id}

                results["phases_completed"] = phase_id + 1

            # Final callback
            if progress_callback:
                progress_callback(11, "Complete", 100.0)

            # Write output if path provided
            if output_path:
                from pathlib import Path
                import json

                output_file = Path(output_path)
                output_file.parent.mkdir(parents=True, exist_ok=True)

                with open(output_file, 'w', encoding='utf-8') as f:
                    json.dump(results, f, indent=2, ensure_ascii=False, default=str)

                logger.info(f"Results written to: {output_path}")
                results["metadata"]["output_path"] = str(output_path)

            results["metadata"]["end_time"] = datetime.now().isoformat()
            results["metadata"]["success"] = True

            root_span.set_attribute("orchestration_success", True)
            logger.info("orchestration_complete",
phases_completed=results["phases_completed"])

            return results

        except Exception as e:
```

```
            logger.error(f"Orchestration failed: {e}", exc_info=True)
            root_span.set_attribute("orchestration_success", False)
            root_span.set_attribute("error", str(e))

            results["metadata"]["end_time"] = datetime.now().isoformat()
            results["metadata"]["success"] = False
            results["metadata"]["error"] = str(e)

            raise RuntimeError(f"Orchestration pipeline failed: {e}") from e

    def execute_sophisticated_engineering_operation(self, policy_area_id: str) ->
dict[str, Any]:
        """
        Orchestrates a sophisticated engineering operation:
        1. Generates 10 smart policy chunks using the canonical SPC ingestion pipeline.
        2. Loads the corresponding signals (patterns and regex) for the policy area.
        3. Instantiates an executor.
        4. Distributes a "work package" (chunks and signals) to the executor.
        5. Returns the generated artifacts as evidence.
        """
        logger.info(f"--- Starting Sophisticated Engineering Operation for:
{policy_area_id} ---")

        # 1. Generate 10 smart policy chunks
        from pathlib import Path

        from saaaaaa.processing.spc_ingestion import CPPIngestionPipeline

        document_path = Path(f"data/policy_areas/{policy_area_id}.txt")
        logger.info(f"Processing document: {document_path}")

        ingestion_pipeline = CPPIngestionPipeline()
        canon_package = asyncio.run(ingestion_pipeline.process(document_path,
max_chunks=10))

        logger.info(f"Generated {len(canon_package.chunk_graph.chunks)} chunks for
{policy_area_id}.")

        # 2. Load signals
        from .questionnaire import load_questionnaire
        from .signal_loader import build_signal_pack_from_monolith

        questionnaire = load_questionnaire()
        signal_pack = build_signal_pack_from_monolith(policy_area_id,
questionnaire=questionnaire)
        logger.info(f"Loaded signal pack for {policy_area_id} with
{len(signal_pack.patterns)} patterns.")

        # 3. Instantiate an executor
        from . import executors

        # Simple mock for the signal registry, as the executor expects an object with a
'get' method.
        class MockSignalRegistry:
            def __init__(self, pack) -> None:
                self._pack = pack
            def get(self, _policy_area):
                return self._pack

        executor_instance = executors.D1Q1_Executor(
            method_executor=self.executor,
            signal_registry=MockSignalRegistry(signal_pack)
        )
        logger.info(f"Instantiated executor: {executor_instance.__class__.__name__}")

        # 4. Prepare and "distribute" the work package
        work_package = {
            "canon_policy_package": canon_package.to_dict(),
```

```
            "signal_pack": signal_pack.to_dict(),
        }

        logger.info(f"Distributing work package to executor for {policy_area_id}.")
        # This simulates the distribution. The executor method will provide the evidence
of receipt.
        if hasattr(executor_instance, 'receive_and_process_work_package'):
            executor_instance.receive_and_process_work_package(work_package)
        else:
            logger.error("Executor does not have the 'receive_and_process_work_package'
method.")

        logger.info(f"--- Completed Sophisticated Engineering Operation for:
{policy_area_id} ---")

        # 5. Return evidence
        return {
            "canon_package": canon_package.to_dict(),
            "signal_pack": signal_pack.to_dict(),
        }

    def _resolve_path(self, path: str | None) -> str | None:
        """Resolve a relative or absolute path, searching multiple candidate locations."""
        if path is None:
            return None
        resolved = resolve_workspace_path(path)
        return str(resolved)

    def _get_phase_timeout(self, phase_id: int) -> float:
        """Get timeout for a specific phase."""
        return self.PHASE_TIMEOUTS.get(phase_id, 300.0)  # Default 5 minutes

    def process_development_plan(
            self, pdf_path: str, preprocessed_document: Any | None = None
    ) -> list[PhaseResult]:
        try:
            loop = asyncio.get_running_loop()
        except RuntimeError:
            loop = None
        if loop and loop.is_running():
            raise RuntimeError("process_development_plan() debe ejecutarse fuera de un
loop asyncio activo")
        return asyncio.run(
            self.process_development_plan_async(
                pdf_path, preprocessed_document=preprocessed_document
            )
        )

    async def process(self, preprocessed_document: Any) -> list[PhaseResult]:
        """
        DEPRECATED ALIAS for process_development_plan_async().

        This method exists ONLY for backward compatibility with code
        that incorrectly assumed Orchestrator had a .process() method.

        Use process_development_plan_async() instead.

        Args:
            preprocessed_document: PreprocessedDocument to process

        Returns:
            List of phase results

        Raises:
            DeprecationWarning: This method is deprecated
        """
        import warnings
        warnings.warn(
```

```
            "Orchestrator.process() is deprecated. "
            "Use process_development_plan_async(pdf_path, preprocessed_document=...)
instead.",
            DeprecationWarning,
            stacklevel=2
        )

        # Extract pdf_path from preprocessed_document if available
        pdf_path = getattr(preprocessed_document, 'source_path', None)
        if pdf_path is None:
            # Try to get from metadata
            metadata = getattr(preprocessed_document, 'metadata', {})
            pdf_path = metadata.get('source_path', 'unknown.pdf')

        return await self.process_development_plan_async(
            pdf_path=str(pdf_path),
            preprocessed_document=preprocessed_document
        )

    async def process_development_plan_async(
            self, pdf_path: str, preprocessed_document: Any | None = None
    ) -> list[PhaseResult]:
        self.reset_abort()
        self.phase_results = []
        self._phase_instrumentation = {}
        self._phase_outputs = {}
        self._context = {"pdf_path": pdf_path}
        if preprocessed_document is not None:
            self._context["preprocessed_override"] = preprocessed_document
        self._phase_status = {phase_id: "not_started" for phase_id, *_ in self.FASES}
        self._start_time = time.perf_counter()

        for phase_id, mode, handler_name, phase_label in self.FASES:
            self._ensure_not_aborted()
            handler = getattr(self, handler_name)
            instrumentation = PhaseInstrumentation(
                phase_id=phase_id,
                name=phase_label,
                items_total=self.PHASE_ITEM_TARGETS.get(phase_id),
                snapshot_interval=self.resource_snapshot_interval,
                resource_limits=self.resource_limits,
            )
            instrumentation.start(items_total=self.PHASE_ITEM_TARGETS.get(phase_id))
            self._phase_instrumentation[phase_id] = instrumentation
            self._phase_status[phase_id] = "running"

            args = [self._context[key] for key in self.PHASE_ARGUMENT_KEYS.get(phase_id,
[])]

            success = False
            data: Any = None
            error: Exception | None = None
            try:
                if mode == "sync":
                    data = handler(*args)
                else:
                    # Use centralized execute_phase_with_timeout
                    data = await execute_phase_with_timeout(
                        phase_id,
                        phase_label,
                        handler,
                        *args,
                        timeout_s=self._get_phase_timeout(phase_id),
                    )
                success = True
            except PhaseTimeoutError as exc:
                error = exc
                success = False
```

```
                instrumentation.record_error("timeout", str(exc))
                self.request_abort(f"Fase {phase_id} timed out: {exc}")
            except AbortRequested as exc:
                error = exc
                success = False
                instrumentation.record_warning("abort", str(exc))
            except Exception as exc:  # pragma: no cover - defensive logging
                logger.exception("Fase %s falló", phase_label)
                error = exc
                success = False
                instrumentation.record_error("exception", str(exc))
                self.request_abort(f"Fase {phase_id} falló: {exc}")
            finally:
                instrumentation.complete()

            aborted = self.abort_signal.is_aborted()
            duration_ms = instrumentation.duration_ms() or 0.0
            phase_result = PhaseResult(
                success=success and not aborted,
                phase_id=str(phase_id),
                data=data,
                error=error,
                duration_ms=duration_ms,
                mode=mode,
                aborted=aborted,
            )
            self.phase_results.append(phase_result)

            if success and not aborted:
                self._phase_outputs[phase_id] = data
                out_key = self.PHASE_OUTPUT_KEYS.get(phase_id)
                if out_key:
                    self._context[out_key] = data
                self._phase_status[phase_id] = "completed"
            elif aborted:
                self._phase_status[phase_id] = "aborted"
                break
            else:
                self._phase_status[phase_id] = "failed"
                break

        return self.phase_results

    def get_processing_status(self) -> dict[str, Any]:
        if self._start_time is None:
            status = "not_started"
            elapsed = 0.0
            completed_flag = False
        else:
            aborted = self.abort_signal.is_aborted()
            status = "aborted" if aborted else "running"
            elapsed = time.perf_counter() - self._start_time
            completed_flag = all(state == "completed" for state in
self._phase_status.values()) and not aborted

        completed = sum(1 for state in self._phase_status.values() if state ==
"completed")
        total = len(self.FASES)
        overall_progress = completed / total if total else 0.0

        phase_progress = {
            str(phase_id): instr.progress()
            for phase_id, instr in self._phase_instrumentation.items()
        }

        resource_usage = self.resource_limits.get_resource_usage() if self._start_time
else {}
```

```
        return {
            "status": status,
            "overall_progress": overall_progress,
            "phase_progress": phase_progress,
            "elapsed_time_s": elapsed,
            "resource_usage": resource_usage,
            "abort_status": self.abort_signal.is_aborted(),
            "abort_reason": self.abort_signal.get_reason(),
            "completed": completed_flag,
        }

    def get_phase_metrics(self) -> dict[str, Any]:
        return {
            str(phase_id): instr.build_metrics()
            for phase_id, instr in self._phase_instrumentation.items()
        }

    async def monitor_progress_async(self, poll_interval: float = 2.0):
        while True:
            status = self.get_processing_status()
            yield status
            if status["status"] != "running":
                break
            await asyncio.sleep(poll_interval)

    def abort_handler(self, reason: str) -> None:
        self.request_abort(reason)

    def request_abort(self, reason: str) -> None:
        """Request orchestration to abort with a specific reason."""
        self.abort_signal.abort(reason)
        logger.warning(f"Abort requested: {reason}")

    def reset_abort(self) -> None:
        """Reset the abort signal to allow new orchestration runs."""
        self.abort_signal.reset()
        logger.debug("Abort signal reset")

    def _ensure_not_aborted(self) -> None:
        """Check if orchestration has been aborted and raise exception if so."""
        if self.abort_signal.is_aborted():
            reason = self.abort_signal.get_reason() or "Unknown reason"
            raise AbortRequested(f"Orchestration aborted: {reason}")

    def health_check(self) -> dict[str, Any]:
        usage = self.resource_limits.get_resource_usage()
        cpu_headroom = max(0.0, self.resource_limits.max_cpu_percent -
usage.get("cpu_percent", 0.0))
        mem_headroom = max(0.0, (self.resource_limits.max_memory_mb or 0.0) -
usage.get("rss_mb", 0.0))
        score = max(0.0, min(100.0, (cpu_headroom / max(1.0,
self.resource_limits.max_cpu_percent)) * 50.0))
        if self.resource_limits.max_memory_mb:
            score += max(0.0, min(50.0, (mem_headroom / max(1.0,
self.resource_limits.max_memory_mb)) * 50.0))
        score = min(100.0, score)
        if self.abort_signal.is_aborted():
            score = min(score, 20.0)
        return {"score": score, "resource_usage": usage, "abort":
self.abort_signal.is_aborted()}

    def get_system_health(self) -> dict[str, Any]:
        """
        Comprehensive system health check.

        Returns health status with component checks for:
        - Method executor
        - Questionnaire provider (if available)
```

```
- Resource limits and usage

Returns:
    Dict with overall status ('healthy', 'degraded', 'unhealthy')
    and component-specific health information
"""
health = {
    'status': 'healthy',
    'timestamp': datetime.utcnow().isoformat(),
    'components': {}
}

# Check method executor
try:
    executor_health = {
        'instances_loaded': len(self.executor.instances),
        'calibrations_loaded': len(self.executor.calibrations),
        'status': 'healthy'
    }
    health['components']['method_executor'] = executor_health
except Exception as e:
    health['status'] = 'unhealthy'
    health['components']['method_executor'] = {
        'status': 'unhealthy',
        'error': str(e)
    }

# Check questionnaire provider (if available)
try:
    from . import get_questionnaire_provider
    provider = get_questionnaire_provider()
    questionnaire_health = {
        'has_data': provider.has_data(),
        'status': 'healthy' if provider.has_data() else 'unhealthy'
    }
    health['components']['questionnaire_provider'] = questionnaire_health

    if not provider.has_data():
        health['status'] = 'degraded'
except Exception as e:
    health['status'] = 'unhealthy'
    health['components']['questionnaire_provider'] = {
        'status': 'unhealthy',
        'error': str(e)
    }

# Check resource limits
try:
    usage = self.resource_limits.get_resource_usage()
    resource_health = {
        'cpu_percent': usage.get('cpu_percent', 0),
        'memory_mb': usage.get('rss_mb', 0),
        'worker_budget': usage.get('worker_budget', 0),
        'status': 'healthy'
    }

    # Warning thresholds
    if usage.get('cpu_percent', 0) > 80:
        resource_health['status'] = 'degraded'
        resource_health['warning'] = 'High CPU usage'
        health['status'] = 'degraded'

    if usage.get('rss_mb', 0) > 3500:  # Near 4GB limit
        resource_health['status'] = 'degraded'
        resource_health['warning'] = 'High memory usage'
        health['status'] = 'degraded'

    health['components']['resources'] = resource_health
```

```
        except Exception as e:
            health['status'] = 'unhealthy'
            health['components']['resources'] = {
                'status': 'unhealthy',
                'error': str(e)
            }

        # Check abort status
        if self.abort_signal.is_aborted():
            health['status'] = 'unhealthy'
            health['abort_reason'] = self.abort_signal.get_reason()

        return health

    def export_metrics(self) -> dict[str, Any]:
        """
        Export all metrics for monitoring.

        Returns:
            Dict containing:
            - timestamp: Current UTC timestamp
            - phase_metrics: Metrics for all phases
            - resource_usage: Resource usage history
            - abort_status: Current abort status
            - phase_status: Status of all phases
        """
        abort_timestamp = self.abort_signal.get_timestamp()

        return {
            'timestamp': datetime.utcnow().isoformat(),
            'phase_metrics': self.get_phase_metrics(),
            'resource_usage': self.resource_limits.get_usage_history(),
            'abort_status': {
                'is_aborted': self.abort_signal.is_aborted(),
                'reason': self.abort_signal.get_reason(),
                'timestamp': abort_timestamp.isoformat() if abort_timestamp else None,
            },
            'phase_status': dict(self._phase_status),
        }

    def _load_configuration(self) -> dict[str, Any]:
        self._ensure_not_aborted()
        instrumentation = self._phase_instrumentation[0]
        start = time.perf_counter()

        # Use pre-loaded monolith data (I/O-free path)
        if self._monolith_data is not None:
            # Normalize monolith for hash and serialization (handles MappingProxyType)
            monolith = _normalize_monolith_for_hash(self._monolith_data)

            # Stable, content-based hash for reproducibility
            monolith_hash = hashlib.sha256(
                json.dumps(monolith, sort_keys=True, ensure_ascii=False, separators=(",",
":")).encode("utf-8")
            ).hexdigest()
        else:
            raise ValueError(
                "No monolith data available. Use saaaaaa.core.orchestrator.factory to load
 "
                "data and pass via monolith parameter for I/O-free initialization."
            )

        micro_questions: list[dict[str, Any]] = monolith["blocks"].get("micro_questions",
[])
        meso_questions: list[dict[str, Any]] = monolith["blocks"].get("meso_questions",
[])
        macro_question: dict[str, Any] = monolith["blocks"].get("macro_question", {})
```

```
        question_total = len(micro_questions) + len(meso_questions) + (1 if macro_question
 else 0)
        if question_total != EXPECTED_QUESTION_COUNT:
            logger.warning("Question count mismatch: expected %s, got %s",
EXPECTED_QUESTION_COUNT, question_total)
            instrumentation.record_error("integrity", f"Conteo de preguntas inesperado:
{question_total}", expected=EXPECTED_QUESTION_COUNT, found=question_total)

        structure_report = self._validate_contract_structure(monolith, instrumentation)

        method_summary: dict[str, Any] = {}
        # Use pre-loaded method_map data (I/O-free path)
        if self._method_map_data is not None:
            method_map = self._method_map_data

            # ========================================================================
            # PROMPT_NONEMPTY_EXECUTION_GRAPH_ENFORCER: Validate method_map is non-empty
            # Cannot route methods with empty map
            # ========================================================================
            if not method_map:
                raise RuntimeError(
                    "Method map is empty - cannot route methods. "
                    "A non-empty method map is required for orchestration."
                )

            summary = method_map.get("summary", {})
            total_methods = summary.get("total_methods")
            if total_methods != EXPECTED_METHOD_COUNT:
                logger.warning("Method count mismatch: expected %s, got %s",
EXPECTED_METHOD_COUNT, total_methods)
                instrumentation.record_error(
                    "catalog",
                    "Total de métodos inesperado",
                    expected=EXPECTED_METHOD_COUNT,
                    found=total_methods,
                )
            method_summary = {
                "total_methods": total_methods,
                "metadata": summary,
            }

        schema_report: dict[str, Any] = {"errors": []}
        # Use pre-loaded schema data (I/O-free path)
        if self._schema_data is not None:
            try:  # pragma: no cover - optional dependency
                import jsonschema

                schema = self._schema_data

                validator = jsonschema.Draft202012Validator(schema)
                schema_errors = [
                    {
                        "path": list(error.path),
                        "message": error.message,
                    }
                    for error in validator.iter_errors(monolith)
                ]
                schema_report["errors"] = schema_errors
                if schema_errors:
                    instrumentation.record_error(
                        "schema",
                        f"Validation errors: {len(schema_errors)}",
                        count=len(schema_errors),
                    )
            except ImportError:
                logger.warning("jsonschema not installed, skipping schema validation")

        duration = time.perf_counter() - start
```

```
        instrumentation.increment(latency=duration)

        aggregation_settings = AggregationSettings.from_monolith(monolith)
        config = {
            "catalog": self.catalog,
            "monolith": monolith,
            "monolith_sha256": monolith_hash,
            "micro_questions": micro_questions,
            "meso_questions": meso_questions,
            "macro_question": macro_question,
            "structure_report": structure_report,
            "method_summary": method_summary,
            "schema_report": schema_report,
            # Internal aggregation settings (underscore denotes private use).
            # Created during Phase 0 as required by the C0-CONFIG-V1.0 contract.
            # Consumed by downstream aggregation logic in later phases.
            "_aggregation_settings": aggregation_settings,
        }

        return config

    def _validate_contract_structure(self, monolith: dict[str, Any], instrumentation:
PhaseInstrumentation) -> dict[
        str, Any]:
        micro_questions = monolith["blocks"].get("micro_questions", [])
        base_slots = {question.get("base_slot") for question in micro_questions}
        modalities = {question.get("scoring_modality") for question in micro_questions}
        expected_modalities = {"TYPE_A", "TYPE_B", "TYPE_C", "TYPE_D", "TYPE_E", "TYPE_F"}

        if len(base_slots) != 30:
            instrumentation.record_error(
                "structure",
                "Cantidad de slots base inválida",
                expected=30,
                found=len(base_slots),
            )

        missing_modalities = expected_modalities - modalities
        if missing_modalities:
            instrumentation.record_error(
                "structure",
                "Modalidades faltantes",
                missing=sorted(missing_modalities),
            )

        slot_area_map: dict[str, str] = {}
        area_cluster_map: dict[str, str] = {}
        for question in micro_questions:
            slot = question.get("base_slot")
            area = question.get("policy_area_id")
            cluster = question.get("cluster_id")
            if slot and area:
                previous = slot_area_map.setdefault(slot, area)
                if previous != area:
                    instrumentation.record_error(
                        "structure",
                        "Asignación de área inconsistente",
                        base_slot=slot,
                        previous=previous,
                        current=area,
                    )
            if area and cluster:
                previous_cluster = area_cluster_map.setdefault(area, cluster)
                if previous_cluster != cluster:
                    instrumentation.record_error(
                        "structure",
                        "Área asignada a múltiples clústeres",
                        area=area,
```

```
                    previous=previous_cluster,
                    current=cluster,
                )

        return {
            "base_slots": sorted(base_slots),
            "modalities": sorted(modalities),
            "slot_area_map": slot_area_map,
            "area_cluster_map": area_cluster_map,
        }

    def _ingest_document(self, pdf_path: str, config: dict[str, Any]) ->
PreprocessedDocument:
        self._ensure_not_aborted()
        instrumentation = self._phase_instrumentation[1]
        start = time.perf_counter()

        document_id = os.path.splitext(os.path.basename(pdf_path))[0] or "doc_1"

        # Initialize and run the canonical SPC ingestion pipeline
        try:
            from saaaaaa.processing.spc_ingestion import CPPIngestionPipeline
            from pathlib import Path

            pipeline = CPPIngestionPipeline()
            # Note: The process method in the pipeline is async
            canon_package = asyncio.run(pipeline.process(
                document_path=Path(pdf_path),
                document_id=document_id
            ))
        except ImportError as e:
            error_msg = f"Failed to import CPPIngestionPipeline: {e}"
            instrumentation.record_error("ingestion", "Import Error", reason=error_msg)
            raise RuntimeError(error_msg) from e
        except Exception as e:
            error_msg = f"SPC Ingestion pipeline failed: {e}"
            instrumentation.record_error("ingestion", "Pipeline Failure",
reason=error_msg)
            raise RuntimeError(error_msg) from e

        # Adapt the output CanonPolicyPackage to the PreprocessedDocument format
        try:
            preprocessed = PreprocessedDocument.ensure(
                canon_package, document_id=document_id, use_spc_ingestion=True
            )
        except (TypeError, ValueError) as exc:
            error_msg = f"Failed to adapt CanonPolicyPackage to PreprocessedDocument:
{exc}"
            instrumentation.record_error(
                "ingestion", "Adapter Error", reason=error_msg
            )
            raise TypeError(error_msg) from exc

        # Validate that the document is not empty
        if not preprocessed.raw_text or not preprocessed.raw_text.strip():
            error_msg = "Empty document after ingestion - raw_text is empty or whitespace-
only"
            instrumentation.record_error(
                "ingestion", "Empty document", reason=error_msg
            )
            raise ValueError(error_msg)

        # === P01-ES v1.0 VALIDATION GATES ===
        # 1. Enforce strict chunk count of 60
        actual_chunk_count = preprocessed.metadata.get("chunk_count", 0)
        if actual_chunk_count != P01_EXPECTED_CHUNK_COUNT:
            error_msg = (
                f"P01 Validation Failed: Expected exactly {P01_EXPECTED_CHUNK_COUNT}
```

```
chunks, "
                f"but found {actual_chunk_count}."
            )
            instrumentation.record_error("ingestion", "Chunk Count Mismatch",
reason=error_msg)
            raise ValueError(error_msg)

        # 2. Enforce presence of policy_area_id and dimension_id in all chunks
        if not preprocessed.chunks:
            error_msg = "P01 Validation Failed: No chunks found in PreprocessedDocument."
            instrumentation.record_error("ingestion", "Empty Chunk List",
reason=error_msg)
            raise ValueError(error_msg)

        for i, chunk in enumerate(preprocessed.chunks):
            # The chunk object from the adapter is a dataclass, so we use getattr
            if not getattr(chunk, 'policy_area_id', None):
                error_msg = f"P01 Validation Failed: Chunk {i} is missing
'policy_area_id'."
                instrumentation.record_error("ingestion", "Missing Metadata",
reason=error_msg)
                raise ValueError(error_msg)
            if not getattr(chunk, 'dimension_id', None):
                error_msg = f"P01 Validation Failed: Chunk {i} is missing 'dimension_id'."
                instrumentation.record_error("ingestion", "Missing Metadata",
reason=error_msg)
                raise ValueError(error_msg)

        logger.info(f"✓ P01-ES v1.0 validation gates passed for {actual_chunk_count}
chunks.")

        text_length = len(preprocessed.raw_text)
        sentence_count = len(preprocessed.sentences) if preprocessed.sentences else 0
        adapter_source = preprocessed.metadata.get("adapter_source", "unknown")

        # Store ingestion information for verification manifest
        ingestion_info = {
            "method": "SPC",  # Only SPC is supported
            "chunk_count": chunk_count,
            "text_length": text_length,
            "sentence_count": sentence_count,
            "adapter_source": adapter_source,
            "chunk_strategy": preprocessed.metadata.get("chunk_strategy", "semantic"),
        }
        if "chunk_overlap" in preprocessed.metadata:
            ingestion_info["chunk_overlap"] = preprocessed.metadata["chunk_overlap"]

        # Store in context for manifest generation
        if hasattr(self, "_context"):
            self._context["ingestion_info"] = ingestion_info

        logger.info(
            f"Document ingested successfully: document_id={document_id}, "
            f"method=SPC, text_length={text_length}, chunk_count={chunk_count}, "
            f"sentence_count={sentence_count}"
        )

        duration = time.perf_counter() - start
        instrumentation.increment(latency=duration)
        return preprocessed

    async def _execute_micro_questions_async(
            self,
            document: PreprocessedDocument,
            config: dict[str, Any],
    ) -> list[MicroQuestionRun]:
        self._ensure_not_aborted()
        instrumentation = self._phase_instrumentation[2]
```

```
micro_questions = config.get("micro_questions", [])
instrumentation.items_total = len(micro_questions)
ordered_questions: list[dict[str, Any]] = []

# NEW: Initialize chunk router for chunk-aware execution
chunk_routes: dict[int, Any] = {}
if document.processing_mode == "chunked" and document.chunks:
    try:
        from saaaaaa.core.orchestrator.chunk_router import ChunkRouter
        router = ChunkRouter()

        # Route chunks to executors
        for chunk in document.chunks:
            route = router.route_chunk(chunk)
            if not route.skip_reason:
                chunk_routes[chunk.id] = route

        logger.info(
            f"Chunk-aware execution enabled: routed {len(chunk_routes)} chunks "
            f"from {len(document.chunks)} total chunks"
        )
    except ImportError:
        logger.warning("ChunkRouter not available, falling back to flat mode")
        chunk_routes = {}

questions_by_slot: dict[str, deque] = {}
for question in micro_questions:
    slot = question.get("base_slot")
    questions_by_slot.setdefault(slot, deque()).append(question)

slots = sorted(questions_by_slot.keys())
while True:
    added = False
    for slot in slots:
        queue = questions_by_slot.get(slot)
        if queue:
            ordered_questions.append(queue.popleft())
            added = True
    if not added:
        break

semaphore = asyncio.Semaphore(self.resource_limits.max_workers)
self.resource_limits.attach_semaphore(semaphore)

circuit_breakers: dict[str, dict[str, Any]] = {
    slot: {"failures": 0, "open": False}
    for slot in self.executors
}

results: list[MicroQuestionRun] = []

# NEW: Track chunk execution metrics
execution_metrics = {
    "chunk_executions": 0,  # Actual chunk-level executions
    "full_doc_executions": 0,  # Fallback full document executions
    "total_chunks_processed": 0,  # Total chunks that could have been processed
}

async def process_question(question: dict[str, Any]) -> MicroQuestionRun:
    await self.resource_limits.apply_worker_budget()
    async with semaphore:
        self._ensure_not_aborted()
        question_id = question.get("question_id", "")
        question_global = int(question.get("question_global", 0))
        base_slot = question.get("base_slot", "")
        metadata = {
            key: question.get(key)
            for key in (
```

```
                    "question_id",
                    "question_global",
                    "base_slot",
                    "dimension_id",
                    "policy_area_id",
                    "cluster_id",
                    "scoring_modality",
                    "expected_elements",
                )
            }

            circuit = circuit_breakers.setdefault(base_slot, {"failures": 0, "open":
False})
            if circuit.get("open"):
                instrumentation.record_warning(
                    "circuit_breaker",
                    "Circuit breaker abierto, pregunta omitida",
                    base_slot=base_slot,
                    question_id=question_id,
                )
                instrumentation.increment()
                return MicroQuestionRun(
                    question_id=question_id,
                    question_global=question_global,
                    base_slot=base_slot,
                    metadata=metadata,
                    evidence=None,
                    error="circuit_breaker_open",
                    aborted=False,
                )

            usage = self.resource_limits.get_resource_usage()
            mem_exceeded, usage = self.resource_limits.check_memory_exceeded(usage)
            cpu_exceeded, usage = self.resource_limits.check_cpu_exceeded(usage)
            if mem_exceeded:
                instrumentation.record_warning("resource", "Límite de memoria
excedido", usage=usage)
            if cpu_exceeded:
                instrumentation.record_warning("resource", "Límite de CPU excedido",
usage=usage)

            executor_class = self.executors.get(base_slot)
            start_time = time.perf_counter()
            evidence: Evidence | None = None
            error_message: str | None = None

            if not executor_class:
                error_message = f"Ejecutor no definido para {base_slot}"
                instrumentation.record_error("executor", error_message,
base_slot=base_slot)
            else:
                try:
                    executor_instance = executor_class(
                        self.executor,
                        signal_registry=self.executor.signal_registry,
                        config=self.executor_config,
                        questionnaire_provider=self.questionnaire_provider,
                        calibration_orchestrator=self.calibration_orchestrator
                    )

                    # Pass the question context to the executor
                    evidence = await asyncio.to_thread(
                        executor_instance.execute, document, self.executor,
question_context=question
                    )
                    circuit["failures"] = 0
                except Exception as exc:  # pragma: no cover - dependencias externas
                    circuit["failures"] += 1
```

```
                    error_message = str(exc)
                    instrumentation.record_error(
                        "micro_question",
                        error_message,
                        base_slot=base_slot,
                        question_id=question_id,
                    )
                    if circuit["failures"] >= 3:
                        circuit["open"] = True
                        instrumentation.record_warning(
                            "circuit_breaker",
                            "Circuit breaker activado",
                            base_slot=base_slot,
                        )

                duration = time.perf_counter() - start_time
                instrumentation.increment(latency=duration)
                if instrumentation.items_processed % 10 == 0:
                    instrumentation.record_warning(
                        "progress",
                        "Progreso de micro preguntas",
                        processed=instrumentation.items_processed,
                        total=instrumentation.items_total,
                    )

                return MicroQuestionRun(
                    question_id=question_id,
                    question_global=question_global,
                    base_slot=base_slot,
                    metadata=metadata,
                    evidence=evidence,
                    error=error_message,
                    duration_ms=duration * 1000.0,
                    aborted=self.abort_signal.is_aborted(),
                )

        tasks = [asyncio.create_task(process_question(question)) for question in
ordered_questions]

        try:
            for task in asyncio.as_completed(tasks):
                result = await task
                results.append(result)
                if self.abort_signal.is_aborted():
                    raise AbortRequested(self.abort_signal.get_reason() or "Abort
requested")
        except AbortRequested:
            for task in tasks:
                task.cancel()
            raise

        # Log chunk execution metrics
        if chunk_routes and document.processing_mode == "chunked":
            total_possible = len(micro_questions) * len(document.chunks)
            actual_executed = execution_metrics["chunk_executions"] +
execution_metrics["full_doc_executions"]
            savings_pct = ((total_possible - actual_executed) / max(total_possible, 1)) *
100 if total_possible > 0 else 0

            logger.info(
                f"Chunk execution metrics: {execution_metrics['chunk_executions']} chunk-
scoped, "
                f"{execution_metrics['full_doc_executions']} full-doc, "
                f"{total_possible} total possible, "
                f"savings: {savings_pct:.1f}%"
            )

            # Store metrics for verification manifest
```

```
            if not hasattr(self, '_execution_metrics'):
                self._execution_metrics = {}
            self._execution_metrics['phase_2'] = {
                'chunk_executions': execution_metrics['chunk_executions'],
                'full_doc_executions': execution_metrics['full_doc_executions'],
                'total_possible_executions': total_possible,
                'actual_executions': actual_executed,
                'savings_percent': savings_pct,
            }

        return results

    async def _score_micro_results_async(
            self,
            micro_results: list[MicroQuestionRun],
            config: dict[str, Any],
    ) -> list[ScoredMicroQuestion]:
        self._ensure_not_aborted()
        instrumentation = self._phase_instrumentation[3]
        instrumentation.items_total = len(micro_results)

        # Import from the flat scoring.py module file
        import importlib.util
        from pathlib import Path
        scoring_file_path = Path(__file__).parent.parent.parent / "analysis" /
"scoring.py"
        spec = importlib.util.spec_from_file_location("scoring_flat", scoring_file_path)
        scoring_flat = importlib.util.module_from_spec(spec)
        spec.loader.exec_module(scoring_flat)
        ScoringEvidence = scoring_flat.Evidence
        MicroQuestionScorer = scoring_flat.MicroQuestionScorer
        ScoringModality = scoring_flat.ScoringModality

        scorer = MicroQuestionScorer()
        results: list[ScoredMicroQuestion] = []
        semaphore = asyncio.Semaphore(self.resource_limits.max_workers)
        self.resource_limits.attach_semaphore(semaphore)

        async def score_item(item: MicroQuestionRun) -> ScoredMicroQuestion:
            async with semaphore:
                await self.resource_limits.apply_worker_budget()
                self._ensure_not_aborted()
                start = time.perf_counter()

                modality_value = item.metadata.get("scoring_modality", "TYPE_A")
                try:
                    modality = ScoringModality(modality_value)
                except Exception:
                    modality = ScoringModality.TYPE_A

                if item.error or not item.evidence:
                    instrumentation.record_warning(
                        "scoring",
                        "Evidencia ausente para scoring",
                        question_id=item.question_id,
                        error=item.error,
                    )
                    instrumentation.increment(latency=time.perf_counter() - start)
                    return ScoredMicroQuestion(
                        question_id=item.question_id,
                        question_global=item.question_global,
                        base_slot=item.base_slot,
                        score=None,
                        normalized_score=None,
                        quality_level=None,
                        evidence=item.evidence,
                        metadata=item.metadata,
                        error=item.error or "missing_evidence",
```

```
            )

        # Handle evidence as either dict or dataclass
        if isinstance(item.evidence, dict):
            elements_found = item.evidence.get("elements", [])
            raw_results = item.evidence.get("raw_results", {})
        else:
            elements_found = getattr(item.evidence, "elements", [])
            raw_results = getattr(item.evidence, "raw_results", {})

        scoring_evidence = ScoringEvidence(
            elements_found=elements_found,
            confidence_scores=raw_results.get("confidence_scores", []),
            semantic_similarity=raw_results.get("semantic_similarity"),
            pattern_matches=raw_results.get("pattern_matches", {}),
            metadata=raw_results,
        )

        try:
            scored = await asyncio.to_thread(
                scorer.apply_scoring_modality,
                item.question_id,
                item.question_global,
                modality,
                scoring_evidence,
            )
            duration = time.perf_counter() - start
            instrumentation.increment(latency=duration)
            return ScoredMicroQuestion(
                question_id=scored.question_id,
                question_global=scored.question_global,
                base_slot=item.base_slot,
                score=scored.raw_score,
                normalized_score=scored.normalized_score,
                quality_level=scored.quality_level.value,
                evidence=item.evidence,
                scoring_details=scored.scoring_details,
                metadata=item.metadata,
            )
        except Exception as exc:  # pragma: no cover - dependencia externa
            instrumentation.record_error(
                "scoring",
                str(exc),
                question_id=item.question_id,
            )
            duration = time.perf_counter() - start
            instrumentation.increment(latency=duration)
            return ScoredMicroQuestion(
                question_id=item.question_id,
                question_global=item.question_global,
                base_slot=item.base_slot,
                score=None,
                normalized_score=None,
                quality_level=None,
                evidence=item.evidence,
                metadata=item.metadata,
                error=str(exc),
            )

tasks = [asyncio.create_task(score_item(item)) for item in micro_results]
for task in asyncio.as_completed(tasks):
    result = await task
    results.append(result)
    if self.abort_signal.is_aborted():
        raise AbortRequested(self.abort_signal.get_reason() or "Abort requested")

return results
```

```
    async def _aggregate_dimensions_async(
        self,
        scored_results: list[ScoredMicroQuestion],
        config: dict[str, Any],
    ) -> list[DimensionScore]:
        """Aggregate micro question scores into dimension scores using
DimensionAggregator.

        Args:
            scored_results: List of scored micro questions
            config: Configuration dict containing monolith

        Returns:
            List of DimensionScore objects with full validation and diagnostics
        """
        self._ensure_not_aborted()
        instrumentation = self._phase_instrumentation[4]

        # Get monolith from config
        monolith = config.get("monolith")
        if not monolith:
            logger.error("No monolith in config for dimension aggregation")
            return []

        aggregation_settings = config.setdefault(
            "_aggregation_settings",
            AggregationSettings.from_monolith(monolith),
        )

        # Initialize dimension aggregator
        aggregator = DimensionAggregator(
            monolith,
            abort_on_insufficient=False,
            aggregation_settings=aggregation_settings,
        )

        scored_payloads: list[dict[str, Any]] = []
        for item in scored_results:
            metadata = item.metadata or {}
            if item.score is None:
                continue
            policy_area = metadata.get("policy_area_id") or metadata.get("policy_area") or
 ""
            dimension = metadata.get("dimension_id") or metadata.get("dimension") or ""
            evidence_payload: dict[str, Any]
            if item.evidence and is_dataclass(item.evidence):
                evidence_payload = asdict(item.evidence)
            elif isinstance(item.evidence, dict):
                evidence_payload = item.evidence
            else:
                evidence_payload = {}
            raw_results = item.scoring_details if isinstance(item.scoring_details, dict)
else {}
            scored_payloads.append(
                {
                    "question_global": item.question_global,
                    "base_slot": item.base_slot,
                    "policy_area": str(policy_area),
                    "dimension": str(dimension),
                    "score": float(item.score),
                    "quality_level": str(item.quality_level or "INSUFICIENTE"),
                    "evidence": evidence_payload,
                    "raw_results": raw_results,
                }
            )

        if not scored_payloads:
            instrumentation.items_total = 0
```

```
            return []

        try:
            validated_results = validate_scored_results(scored_payloads)
        except ValidationError as exc:
            logger.error("Invalid scored results for dimension aggregation: %s", exc)
            raise

        group_by_keys = aggregator.dimension_group_by_keys
        key_func = lambda result: tuple(getattr(result, key, None) for key in
group_by_keys)
        grouped_results = group_by(validated_results, key_func)

        instrumentation.items_total = len(grouped_results)
        dimension_scores: list[DimensionScore] = []

        for group_key, items in grouped_results.items():
            self._ensure_not_aborted()
            await asyncio.sleep(0)
            start = time.perf_counter()
            group_by_values = dict(zip(group_by_keys, group_key, strict=False))
            try:
                dim_score = aggregator.aggregate_dimension(
                    scored_results=items,
                    group_by_values=group_by_values,
                )
                dimension_scores.append(dim_score)
            except Exception as exc:
                logger.error(
                    "Failed to aggregate dimension %s/%s: %s",
                    group_by_values.get("dimension"),
                    group_by_values.get("policy_area"),
                    exc,
                )
            instrumentation.increment(latency=time.perf_counter() - start)

        return dimension_scores

    async def _aggregate_policy_areas_async(
            self,
            dimension_scores: list[DimensionScore],
            config: dict[str, Any],
    ) -> list[AreaScore]:
        """Aggregate dimension scores into policy area scores using AreaPolicyAggregator.

        Args:
            dimension_scores: List of DimensionScore objects
            config: Configuration dict containing monolith

        Returns:
            List of AreaScore objects with full validation and diagnostics
        """
        self._ensure_not_aborted()
        instrumentation = self._phase_instrumentation[5]

        # Get monolith from config
        monolith = config.get("monolith")
        if not monolith:
            logger.error("No monolith in config for area aggregation")
            return []

        aggregation_settings = config.setdefault(
            "_aggregation_settings",
            AggregationSettings.from_monolith(monolith),
        )

        # Initialize area aggregator
        aggregator = AreaPolicyAggregator(
```

```
        monolith,
        abort_on_insufficient=False,
        aggregation_settings=aggregation_settings,
    )

    group_by_keys = aggregator.area_group_by_keys
    key_func = lambda score: tuple(getattr(score, key, None) for key in group_by_keys)
    grouped_scores = group_by(dimension_scores, key_func)

    instrumentation.items_total = len(grouped_scores)
    area_scores: list[AreaScore] = []

    for group_key, scores in grouped_scores.items():
        self._ensure_not_aborted()
        await asyncio.sleep(0)
        start = time.perf_counter()
        group_by_values = dict(zip(group_by_keys, group_key, strict=False))
        try:
            area_score = aggregator.aggregate_area(
                dimension_scores=scores,
                group_by_values=group_by_values,
            )
            area_scores.append(area_score)
        except Exception as exc:
            logger.error(
                "Failed to aggregate policy area %s: %s",
                group_by_values.get("area_id"),
                exc,
            )
        instrumentation.increment(latency=time.perf_counter() - start)

    return area_scores

def _aggregate_clusters(
        self,
        policy_area_scores: list[AreaScore],
        config: dict[str, Any],
) -> list[ClusterScore]:
    """Aggregate policy area scores into cluster scores using ClusterAggregator.

    Args:
        policy_area_scores: List of AreaScore objects
        config: Configuration dict containing monolith

    Returns:
        List of ClusterScore objects with full validation and diagnostics
    """
    self._ensure_not_aborted()
    instrumentation = self._phase_instrumentation[6]

    # Get monolith from config
    monolith = config.get("monolith")
    if not monolith:
        logger.error("No monolith in config for cluster aggregation")
        return []

    aggregation_settings = config.setdefault(
        "_aggregation_settings",
        AggregationSettings.from_monolith(monolith),
    )

    # Initialize cluster aggregator
    aggregator = ClusterAggregator(
        monolith,
        abort_on_insufficient=False,
        aggregation_settings=aggregation_settings,
    )
```

```
        clusters = monolith["blocks"]["niveles_abstraccion"]["clusters"]

        area_to_cluster: dict[str, str] = {}
        for cluster in clusters:
            cluster_id = cluster.get("cluster_id")
            for area_id in cluster.get("policy_area_ids", []):
                if cluster_id and area_id:
                    area_to_cluster[area_id] = cluster_id

        enriched_scores: list[AreaScore] = []
        for score in policy_area_scores:
            cluster_id = area_to_cluster.get(score.area_id)
            if not cluster_id:
                logger.warning(
                    "Area %s not mapped to any cluster definition",
                    score.area_id,
                )
                continue
            score.cluster_id = cluster_id
            enriched_scores.append(score)

        group_by_keys = aggregator.cluster_group_by_keys
        key_func = lambda area_score: tuple(getattr(area_score, key, None) for key in
group_by_keys)
        grouped_scores = group_by(enriched_scores, key_func)

        instrumentation.items_total = len(grouped_scores)
        cluster_scores: list[ClusterScore] = []

        for group_key, scores in grouped_scores.items():
            self._ensure_not_aborted()
            start = time.perf_counter()
            group_by_values = dict(zip(group_by_keys, group_key, strict=False))
            try:
                cluster_score = aggregator.aggregate_cluster(
                    area_scores=scores,
                    group_by_values=group_by_values,
                )
                cluster_scores.append(cluster_score)
            except Exception as exc:
                logger.error(
                    "Failed to aggregate cluster %s: %s",
                    group_by_values.get("cluster_id"),
                    exc,
                )
            instrumentation.increment(latency=time.perf_counter() - start)

        return cluster_scores

    def _evaluate_macro(self, cluster_scores: list[ClusterScore], config: dict[str, Any])
-> MacroScoreDict:
        """Evaluate macro level using MacroAggregator.

        Args:
            cluster_scores: List of ClusterScore objects from FASE 6
            config: Configuration dict containing monolith

        Returns:
            MacroScoreDict with macro_score, macro_score_normalized, and cluster_scores
        """
        self._ensure_not_aborted()
        instrumentation = self._phase_instrumentation[7]
        start = time.perf_counter()

        # Get monolith from config
        monolith = config.get("monolith")
        if not monolith:
            logger.error("No monolith in config for macro evaluation")
```

```
    macro_score = MacroScore(
        score=0.0,
        quality_level="INSUFICIENTE",
        cross_cutting_coherence=0.0,
        systemic_gaps=[],
        strategic_alignment=0.0,
        cluster_scores=[],
        validation_passed=False,
        validation_details={"error": "No monolith", "type": "config"}
    )
    result: MacroScoreDict = {
        "macro_score": macro_score,
        "macro_score_normalized": 0.0,
        "cluster_scores": cluster_scores,
        "cross_cutting_coherence": macro_score.cross_cutting_coherence,
        "systemic_gaps": macro_score.systemic_gaps,
        "strategic_alignment": macro_score.strategic_alignment,
        "quality_band": macro_score.quality_level,
    }
    return result

aggregation_settings = config.setdefault(
    "_aggregation_settings",
    AggregationSettings.from_monolith(monolith),
)

# Initialize macro aggregator
aggregator = MacroAggregator(
    monolith,
    abort_on_insufficient=False,
    aggregation_settings=aggregation_settings,
)

# Extract area_scores and dimension_scores from cluster_scores
area_scores: list[AreaScore] = []
dimension_scores: list[DimensionScore] = []

for cluster in cluster_scores:
    area_scores.extend(cluster.area_scores)
    for area in cluster.area_scores:
        dimension_scores.extend(area.dimension_scores)

# Remove duplicates (in case areas appear in multiple clusters)
seen_areas = set()
unique_areas = []
for area in area_scores:
    if area.area_id not in seen_areas:
        seen_areas.add(area.area_id)
        unique_areas.append(area)

seen_dims = set()
unique_dims = []
for dim in dimension_scores:
    key = (dim.dimension_id, dim.area_id)
    if key not in seen_dims:
        seen_dims.add(key)
        unique_dims.append(dim)

# Evaluate macro
try:
    macro_score = aggregator.evaluate_macro(
        cluster_scores=cluster_scores,
        area_scores=unique_areas,
        dimension_scores=unique_dims
    )
except Exception as e:
    logger.error(f"Failed to evaluate macro: {e}")
    macro_score = MacroScore(
```

```
                score=0.0,
                quality_level="INSUFICIENTE",
                cross_cutting_coherence=0.0,
                systemic_gaps=[],
                strategic_alignment=0.0,
                cluster_scores=cluster_scores,
                validation_passed=False,
                validation_details={"error": str(e), "type": "exception"}
            )

        instrumentation.increment(latency=time.perf_counter() - start)
        # macro_score is already normalized to 0-1 range from averaging cluster scores
        # Extract the score field from the MacroScore object with explicit float
conversion
        macro_score_normalized = float(macro_score.score) if isinstance(macro_score,
MacroScore) else float(macro_score)

        result: MacroScoreDict = {
            "macro_score": macro_score,
            "macro_score_normalized": macro_score_normalized,
            "cluster_scores": cluster_scores,
            "cross_cutting_coherence": macro_score.cross_cutting_coherence,
            "systemic_gaps": macro_score.systemic_gaps,
            "strategic_alignment": macro_score.strategic_alignment,
            "quality_band": macro_score.quality_level,
        }
        return result

    async def _generate_recommendations(
            self,
            macro_result: dict[str, Any],
            config: dict[str, Any],
    ) -> dict[str, Any]:
        """
        Generate recommendations at MICRO, MESO, and MACRO levels using
RecommendationEngine.

        This phase connects to the orchestrator's 3-level flux:
        - MICRO: Uses scored question results from phase 3
        - MESO: Uses cluster aggregations from phase 6
        - MACRO: Uses macro evaluation from phase 7

        Args:
            macro_result: Macro evaluation results from phase 7
            config: Configuration dictionary

        Returns:
            Dictionary with MICRO, MESO, and MACRO recommendations
        """
        self._ensure_not_aborted()
        instrumentation = self._phase_instrumentation[8]
        start = time.perf_counter()

        await asyncio.sleep(0)

        # If RecommendationEngine is not available, return empty recommendations
        if self.recommendation_engine is None:
            logger.warning("RecommendationEngine not available, returning empty
recommendations")
            recommendations = {
                "MICRO": {"level": "MICRO", "recommendations": [], "generated_at":
datetime.utcnow().isoformat()},
                "MESO": {"level": "MESO", "recommendations": [], "generated_at":
datetime.utcnow().isoformat()},
                "MACRO": {"level": "MACRO", "recommendations": [], "generated_at":
datetime.utcnow().isoformat()},
                "macro_score": macro_result.get("macro_score"),
            }
```

```
            instrumentation.increment(latency=time.perf_counter() - start)
            return recommendations

        try:
            # ========================================================================
            # MICRO LEVEL: Transform scored results to PA-DIM scores
            # ========================================================================
            micro_scores: dict[str, float] = {}
            scored_results = self._context.get('scored_results', [])

            # Group by policy area and dimension to calculate average scores
            pa_dim_groups: dict[str, list[float]] = {}
            for result in scored_results:
                if hasattr(result, 'metadata') and result.metadata:
                    pa_id = result.metadata.get('policy_area_id')
                    dim_id = result.metadata.get('dimension_id')
                    score = result.normalized_score

                    if pa_id and dim_id and score is not None:
                        key = f"{pa_id}-{dim_id}"
                        if key not in pa_dim_groups:
                            pa_dim_groups[key] = []
                        pa_dim_groups[key].append(score)

            # Calculate average for each PA-DIM combination
            for key, scores in pa_dim_groups.items():
                if scores:
                    micro_scores[key] = sum(scores) / len(scores)

            logger.info(f"Extracted {len(micro_scores)} MICRO PA-DIM scores for
recommendations")

            # ========================================================================
            # MESO LEVEL: Transform cluster scores
            # ========================================================================
            cluster_data: dict[str, Any] = {}
            cluster_scores = self._context.get('cluster_scores', [])

            for cluster in cluster_scores:
                cluster_id = cluster.get('cluster_id')
                cluster_score = cluster.get('score')
                areas = cluster.get('areas', [])

                if cluster_id and cluster_score is not None:
                    # cluster_score is already normalized to 0-1 range from aggregation
                    normalized_cluster_score = cluster_score

                    # Calculate variance across areas in this cluster using normalized
scores
                    # area scores are already normalized to 0-1 range from aggregation
                    valid_area_scores = [
                        (area, area.get('score'))
                        for area in areas
                        if area.get('score') is not None
                    ]
                    normalized_area_values = [score for _, score in valid_area_scores]
                    variance = (
                        statistics.variance(normalized_area_values)
                        if len(normalized_area_values) > 1
                        else 0.0
                    )

                    # Find weakest policy area in cluster
                    weakest_area = (
                        min(valid_area_scores, key=lambda item: item[1])
                        if valid_area_scores
                        else None
                    )
```

```
                weak_pa = weakest_area[0].get('area_id') if weakest_area else None

                cluster_data[cluster_id] = {
                    'score': normalized_cluster_score * self.PERCENTAGE_SCALE,  #
0-100 scale
                    'variance': variance,
                    'weak_pa': weak_pa
                }

        logger.info(f"Extracted {len(cluster_data)} MESO cluster metrics for
recommendations")

        # ============================================================================
        # MACRO LEVEL: Transform macro evaluation
        # ============================================================================
        macro_score = macro_result.get('macro_score')
        macro_score_normalized = macro_result.get('macro_score_normalized')

        # macro_score is already normalized to 0-1 range
        # Extract the score value if macro_score is a MacroScore object
        if macro_score is not None and macro_score_normalized is None:
            macro_score_normalized = macro_score.score if isinstance(macro_score,
MacroScore) else macro_score

        # Extract numeric value from macro_score_normalized (may be dict/object)
        macro_score_numeric = None
        if macro_score_normalized is not None:
            if isinstance(macro_score_normalized, dict):
                macro_score_numeric = macro_score_normalized.get('score')
            elif hasattr(macro_score_normalized, 'score'):
                try:
                    macro_score_numeric = macro_score_normalized.score
                except (AttributeError, TypeError) as e:
                    logger.warning(f"Failed to extract score attribute: {e}")
                    macro_score_numeric = None
            else:
                # Already a numeric value
                macro_score_numeric = macro_score_normalized

            # Validate that extracted value is numeric
            if macro_score_numeric is not None and not isinstance(macro_score_numeric,
 (int, float)):
                logger.warning(
                    f"Expected numeric macro_score, got
{type(macro_score_numeric).__name__}: {macro_score_numeric!r}"
                )
                macro_score_numeric = None

        # Determine macro band based on score
        macro_band = 'INSUFICIENTE'
        if macro_score_numeric is not None:
            scaled_score = float(macro_score_numeric) * self.PERCENTAGE_SCALE
            if scaled_score >= 75:
                macro_band = 'SATISFACTORIO'
            elif scaled_score >= 55:
                macro_band = 'ACEPTABLE'
            elif scaled_score >= 35:
                macro_band = 'DEFICIENTE'

        # Find clusters below target (< 55%)
        # cluster scores are already normalized to 0-1 range
        clusters_below_target = []
        for cluster in cluster_scores:
            cluster_id = cluster.get('cluster_id')
            cluster_score = cluster.get('score', 0)
            if cluster_score is not None and cluster_score * self.PERCENTAGE_SCALE <
55:
                clusters_below_target.append(cluster_id)
```

```
        # Calculate overall variance
        # cluster scores are already normalized to 0-1 range
        normalized_cluster_scores = [
            c.get('score')
            for c in cluster_scores
            if c.get('score') is not None
        ]
        overall_variance = (
            statistics.variance(normalized_cluster_scores)
            if len(normalized_cluster_scores) > 1
            else 0.0
        )

        variance_alert = 'BAJA'
        if overall_variance >= 0.18:
            variance_alert = 'ALTA'
        elif overall_variance >= 0.08:
            variance_alert = 'MODERADA'

        # Find priority micro gaps (lowest scoring PA-DIM combinations)
        sorted_micro = sorted(micro_scores.items(), key=lambda x: x[1])
        priority_micro_gaps = [k for k, v in sorted_micro[:5] if v < 0.55]

        macro_data = {
            'macro_band': macro_band,
            'clusters_below_target': clusters_below_target,
            'variance_alert': variance_alert,
            'priority_micro_gaps': priority_micro_gaps,
            'macro_score_percentage': (
                float(macro_score_numeric) * self.PERCENTAGE_SCALE if
macro_score_numeric is not None else None
            )
        }

        logger.info(f"Macro band: {macro_band}, Clusters below target:
{len(clusters_below_target)}")

        # ========================================================================
        # GENERATE RECOMMENDATIONS AT ALL 3 LEVELS
        # ========================================================================
        context = {
            'generated_at': datetime.utcnow().isoformat(),
            'macro_score': macro_score
        }

        recommendation_sets = self.recommendation_engine.generate_all_recommendations(
            micro_scores=micro_scores,
            cluster_data=cluster_data,
            macro_data=macro_data,
            context=context
        )

        # Convert RecommendationSet objects to dictionaries
        recommendations = {
            level: rec_set.to_dict() for level, rec_set in recommendation_sets.items()
        }
        recommendations['macro_score'] = macro_score
        recommendations['macro_score_normalized'] = macro_score_normalized

        logger.info(
            f"Generated recommendations: "
            f"MICRO={len(recommendation_sets['MICRO'].recommendations)}, "
            f"MESO={len(recommendation_sets['MESO'].recommendations)}, "
            f"MACRO={len(recommendation_sets['MACRO'].recommendations)}"
        )

    except Exception as e:
```

```
            logger.error(f"Error generating recommendations: {e}", exc_info=True)
            recommendations = {
                "MICRO": {"level": "MICRO", "recommendations": [], "generated_at":
datetime.utcnow().isoformat()},
                "MESO": {"level": "MESO", "recommendations": [], "generated_at":
datetime.utcnow().isoformat()},
                "MACRO": {"level": "MACRO", "recommendations": [], "generated_at":
datetime.utcnow().isoformat()},
                "macro_score": macro_result.get("macro_score"),
                "error": str(e)
            }

        instrumentation.increment(latency=time.perf_counter() - start)
        return recommendations

    def _assemble_report(self, recommendations: dict[str, Any], config: dict[str, Any]) ->
 dict[str, Any]:
        self._ensure_not_aborted()
        instrumentation = self._phase_instrumentation[9]
        start = time.perf_counter()

        report = {
            "generated_at": datetime.utcnow().isoformat(),
            "recommendations": recommendations,
            "metadata": {
                "monolith_sha256": config.get("monolith_sha256"),
                "method_summary": config.get("method_summary"),
            },
        }

        instrumentation.increment(latency=time.perf_counter() - start)
        return report

    async def _format_and_export(self, report: dict[str, Any], config: dict[str, Any]) ->
dict[str, Any]:
        self._ensure_not_aborted()
        instrumentation = self._phase_instrumentation[10]
        start = time.perf_counter()

        await asyncio.sleep(0)
        export_payload = {
            "report": report,
            "phase_metrics": self.get_phase_metrics(),
            "completed_at": datetime.utcnow().isoformat(),
        }

        instrumentation.increment(latency=time.perf_counter() - start)
        return export_payload


def describe_pipeline_shape(
    monolith: dict[str, Any] | None = None,
    executor_instances: dict[str, Any] | None = None,
) -> dict[str, Any]:
    """Describe the actual pipeline shape from live data.

    Computes phase count, question count, and executor count from real data
    instead of using hard-coded constants.

    Args:
        monolith: Questionnaire monolith (if available)
        executor_instances: MethodExecutor.instances dict (if available)

    Returns:
        Dict with actual pipeline metrics
    """
    shape: dict[str, Any] = {
        "phases": len(Orchestrator.FASES),
```

```
    }

    if monolith:
        micro_questions = monolith.get("blocks", {}).get("micro_questions", [])
        meso_questions = monolith.get("blocks", {}).get("meso_questions", [])
        macro_question = monolith.get("blocks", {}).get("macro_question", {})
        question_total = len(micro_questions) + len(meso_questions) + (1 if macro_question
 else 0)
        shape["expected_micro_questions"] = question_total

    if executor_instances:
        shape["registered_executors"] = len(executor_instances)

    return shape

===== FILE: src/saaaaaa/core/orchestrator/evidence_assembler.py =====
from __future__ import annotations

import statistics
from typing import Any, Iterable, Literal

try:
    import structlog
    logger = structlog.get_logger(__name__)
except ImportError:
    import logging
    logger = logging.getLogger(__name__)

def _resolve_value(source: str, method_outputs: dict[str, Any]) -> Any:
    """Resolve dotted source paths from method_outputs."""
    if not source:
        return None
    parts = source.split(".")
    current: Any = method_outputs
    for idx, part in enumerate(parts):
        if idx == 0 and part in method_outputs:
            current = method_outputs[part]
            continue
        if isinstance(current, dict) and part in current:
            current = current[part]
        else:
            return None
    return current


class EvidenceAssembler:
    """
    Assemble evidence fields from method outputs using deterministic merge strategies.
    """

    MERGE_STRATEGIES = {
        "concat",
        "first",
        "last",
        "mean",
        "max",
        "min",
        "weighted_mean",
        "majority",
    }

    @staticmethod
    def assemble(method_outputs: dict[str, Any], assembly_rules: list[dict[str, Any]]) ->
dict[str, Any]:
        evidence: dict[str, Any] = {}
        trace: dict[str, Any] = {}

        if "_signal_usage" in method_outputs:
```

```
            logger.info("signal_consumption_trace",
signals_used=method_outputs["_signal_usage"])
            trace["signal_usage"] = method_outputs["_signal_usage"]
            # Remove from method_outputs to not interfere with evidence assembly
            del method_outputs["_signal_usage"]

        for rule in assembly_rules:
            target = rule.get("target")
            sources: Iterable[str] = rule.get("sources", [])
            strategy: str = rule.get("merge_strategy", "first")
            weights: list[float] | None = rule.get("weights")
            default = rule.get("default")

            if strategy not in EvidenceAssembler.MERGE_STRATEGIES:
                raise ValueError(f"Unsupported merge_strategy '{strategy}' for target
'{target}'")

            values = []
            for src in sources:
                val = _resolve_value(src, method_outputs)
                if val is not None:
                    values.append(val)

            merged = EvidenceAssembler._merge(values, strategy, weights, default)
            evidence[target] = merged
            trace[target] = {"sources": list(sources), "strategy": strategy, "values":
values}

        return {"evidence": evidence, "trace": trace}

    @staticmethod
    def _merge(values: list[Any], strategy: str, weights: list[float] | None, default:
Any) -> Any:
        if not values:
            return default
        if strategy == "first":
            return values[0]
        if strategy == "last":
            return values[-1]
        if strategy == "concat":
            merged: list[Any] = []
            for v in values:
                if isinstance(v, list):
                    merged.extend(v)
                else:
                    merged.append(v)
            return merged
        numeric_values = [float(v) for v in values if EvidenceAssembler._is_number(v)]
        if strategy == "mean":
            return statistics.fmean(numeric_values) if numeric_values else default
        if strategy == "max":
            return max(numeric_values) if numeric_values else default
        if strategy == "min":
            return min(numeric_values) if numeric_values else default
        if strategy == "weighted_mean":
            if not numeric_values:
                return default
            if not weights:
                weights = [1.0] * len(numeric_values)
            w = weights[: len(numeric_values)] or [1.0] * len(numeric_values)
            total = sum(w) or 1.0
            return sum(v * w_i for v, w_i in zip(numeric_values, w)) / total
        if strategy == "majority":
            counts: dict[Any, int] = {}
            for v in values:
                counts[v] = counts.get(v, 0) + 1
            return max(counts.items(), key=lambda item: item[1])[0] if counts else default
        return default
```

```
    @staticmethod
    def _is_number(value: Any) -> bool:
        try:
            float(value)
            return not isinstance(value, bool)
        except (TypeError, ValueError):
            return False

===== FILE: src/saaaaaa/core/orchestrator/evidence_registry.py =====
"""
Evidence Registry: Append-Only JSONL Store with Hash Chain and Provenance DAG Export

This module implements a comprehensive evidence tracking system that:
1. Stores all evidence in append-only JSONL format for immutability
2. Maintains hash-based indexing for fast evidence lookup
3. Implements blockchain-style hash chaining for ledger integrity
4. Exports provenance DAG showing evidence lineage and dependencies
5. Provides cryptographic verification of evidence integrity

Architecture:
- JSONL Storage: One JSON object per line, append-only for audit trail
- Hash Index: SHA-256 hashes for content-addressable storage
- Hash Chain: Each entry links to previous via previous_hash and entry_hash
- Provenance DAG: Directed acyclic graph of evidence dependencies
- Verification: Cryptographic chain-of-custody validation with chain linkage checks

Hash Chain Security:
The registry implements a blockchain-style hash chain where each entry contains:
- content_hash: SHA-256 of the payload (for content verification)
- previous_hash: Hash of the previous entry's entry_hash (creates the chain)
- entry_hash: SHA-256 of (content_hash + previous_hash + metadata)

This ensures that:
1. Any tampering with payload is detected via content_hash mismatch
2. Any tampering with previous_hash is detected via chain verification
3. Entries cannot be reordered without breaking the chain
4. The entire ledger history can be cryptographically verified
"""

from __future__ import annotations

import hashlib
import json
import logging
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__)

@dataclass
class EvidenceRecord:
    """
    Immutable evidence record with provenance metadata and hash chain linkage.

    Each evidence record captures:
    - Unique identifier (hash-based)
    - Evidence payload (method result, analysis output, etc.)
    - Provenance metadata (source, dependencies, lineage)
    - Temporal metadata (timestamp, execution time)
    - Verification data (content hash, chain hashes)

    Hash Chain Fields:
    - content_hash: SHA-256 of payload (verifies content integrity)
```

```
    - previous_hash: entry_hash of previous record (creates chain linkage)
    - entry_hash: SHA-256 of (content + previous_hash + metadata) (unique entry ID)

    The hash chain ensures that:
    1. Tampering with payload breaks content_hash
    2. Tampering with previous_hash breaks chain verification
    3. Entire ledger history is cryptographically verifiable
    """

    # Identification
    evidence_id: str  # SHA-256 hash of content
    evidence_type: str  # Type of evidence (e.g., "method_result", "analysis",
"extraction")

    # Payload
    payload: dict[str, Any]

    # Provenance
    source_method: str | None = None  # FQN of method that produced this evidence
    parent_evidence_ids: list[str] = field(default_factory=list)  # Dependencies
    question_id: str | None = None
    document_id: str | None = None

    # Temporal
    timestamp: float = field(default_factory=time.time)
    execution_time_ms: float = 0.0

    # Verification
    content_hash: str | None = None  # Hash of payload for verification
    previous_hash: str | None = None  # Hash of previous entry in chain (for ledger
integrity)
    entry_hash: str | None = None  # Hash of this entire entry including previous_hash

    # Metadata
    metadata: dict[str, Any] = field(default_factory=dict)

    def __post_init__(self):
        """Generate content hash and entry hash if not provided."""
        if self.content_hash is None:
            self.content_hash = self._compute_content_hash()
        if self.entry_hash is None:
            self.entry_hash = self._compute_entry_hash()

    def _canonical_dump(self, obj: Any) -> str:
        """
        Create canonical JSON representation for deterministic hashing.

        This method ensures:
        - Keys are sorted alphabetically
        - No whitespace in output
        - Consistent handling of None, booleans, numbers
        - Deterministic ordering for nested structures
        - Unicode normalization

        Uses a custom JSON serialization handler to support non-standard types
        commonly found in evidence payloads (dataclasses, NumPy arrays, custom objects).
        The handler converts objects to dicts via __dict__ or falls back to string
        representation, ensuring all evidence can be serialized without exceptions.

        Args:
            obj: Object to serialize

        Returns:
            Canonical JSON string
        """
        # Use separators with no spaces and sort keys for determinism
        # ensure_ascii=True ensures consistent output across platforms
        # Custom handler for non-serializable types (dataclasses, NumPy arrays, etc.)
```

```
    def default_handler(o):
        if hasattr(o, '__dict__'):
            return o.__dict__
        return str(o)

    return json.dumps(
        obj,
        sort_keys=True,
        separators=(',', ':'),
        ensure_ascii=True,
        default=default_handler
    )

def _compute_content_hash(self) -> str:
    """
    Compute SHA-256 hash of payload for content-addressable storage.

    Uses canonical JSON serialization to ensure deterministic hashing
    across different Python versions and platforms.

    Returns:
        Hex digest of SHA-256 hash
    """
    # Create deterministic JSON representation using canonical dump
    payload_json = self._canonical_dump(self.payload)

    # Compute SHA-256 hash
    hash_obj = hashlib.sha256(payload_json.encode('utf-8'))
    return hash_obj.hexdigest()

def _compute_entry_hash(self) -> str:
    """
    Compute SHA-256 hash of the entire entry including previous_hash.
    This creates the hash chain linking entries together.

    Uses canonical JSON serialization for deterministic hashing.

    Returns:
        Hex digest of SHA-256 hash
    """
    # Combine content hash with previous hash to create chain
    # Use empty string for first entry (no predecessor)
    chain_data = {
        'content_hash': self.content_hash,
        'previous_hash': self.previous_hash if self.previous_hash is not None else '',
        'evidence_type': self.evidence_type,
        'timestamp': self.timestamp,
    }
    chain_json = self._canonical_dump(chain_data)
    hash_obj = hashlib.sha256(chain_json.encode('utf-8'))
    return hash_obj.hexdigest()

def verify_integrity(self, previous_record: EvidenceRecord | None = None) -> bool:
    """
    Verify evidence integrity by recomputing hashes and checking chain linkage.

    Args:
        previous_record: The record that should precede this one in the chain

    Returns:
        True if all integrity checks pass, False otherwise
    """
    # Verify content hash matches
    current_content_hash = self._compute_content_hash()
    if current_content_hash != self.content_hash:
        return False

    # Verify entry hash matches
```

```
        current_entry_hash = self._compute_entry_hash()
        if current_entry_hash != self.entry_hash:
            return False

        # If previous record is provided, verify the chain linkage
        if previous_record is not None:
            # Verify that our previous_hash matches the actual hash of the previous record
            if self.previous_hash != previous_record.entry_hash:
                return False

        return True

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

    @classmethod
    def from_dict(cls, data: dict[str, Any]) -> EvidenceRecord:
        """Create evidence record from dictionary."""
        return cls(**data)

    @classmethod
    def create(
        cls,
        evidence_type: str,
        payload: dict[str, Any],
        source_method: str | None = None,
        parent_evidence_ids: list[str] | None = None,
        question_id: str | None = None,
        document_id: str | None = None,
        execution_time_ms: float = 0.0,
        metadata: dict[str, Any] | None = None,
        previous_hash: str | None = None,
    ) -> EvidenceRecord:
        """
        Create a new evidence record with proper hash computation.

        This factory method ensures:
        - Proper initialization order
        - Deterministic hash computation
        - Validation of required fields

        Args:
            evidence_type: Type of evidence
            payload: Evidence data (must be JSON-serializable)
            source_method: FQN of method that produced evidence
            parent_evidence_ids: List of parent evidence IDs
            question_id: Question ID this evidence relates to
            document_id: Document ID this evidence relates to
            execution_time_ms: Execution time in milliseconds
            metadata: Additional metadata
            previous_hash: Hash of previous entry in chain (for chain linkage)

        Returns:
            New EvidenceRecord instance

        Raises:
            ValueError: If required fields are invalid or payload is not serializable
        """
        if not evidence_type:
            raise ValueError("evidence_type is required")

        if not isinstance(payload, dict):
            raise ValueError("payload must be a dictionary")

        # Test that payload is JSON-serializable
        try:
            json.dumps(payload)
```

```
        except (TypeError, ValueError) as e:
            raise ValueError(f"payload must be JSON-serializable: {e}")

        # Create record with temporary evidence_id
        record = cls(
            evidence_id="",
            evidence_type=evidence_type,
            payload=payload,
            source_method=source_method,
            parent_evidence_ids=parent_evidence_ids or [],
            question_id=question_id,
            document_id=document_id,
            execution_time_ms=execution_time_ms,
            metadata=metadata or {},
            previous_hash=previous_hash,
        )

        # Set evidence_id to content hash (computed in __post_init__)
        record.evidence_id = record.content_hash or ""

        return record

@dataclass
class ProvenanceNode:
    """Node in provenance DAG."""

    evidence_id: str
    evidence_type: str
    source_method: str | None
    timestamp: float
    children: list[str] = field(default_factory=list)  # Evidence IDs that depend on this
    parents: list[str] = field(default_factory=list)   # Evidence IDs this depends on

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

@dataclass
class ProvenanceDAG:
    """
    Directed Acyclic Graph of evidence provenance.

    Captures the full lineage of evidence:
    - Which evidence produced which other evidence
    - Method invocation chains
    - Data flow dependencies
    """

    nodes: dict[str, ProvenanceNode] = field(default_factory=dict)

    # Index for fast queries
    by_method: dict[str, list[str]] = field(default_factory=lambda: defaultdict(list))
    by_type: dict[str, list[str]] = field(default_factory=lambda: defaultdict(list))
    by_question: dict[str, list[str]] = field(default_factory=lambda: defaultdict(list))

    def add_evidence(
        self,
        evidence: EvidenceRecord
    ) -> None:
        """
        Add evidence to provenance DAG.

        Args:
            evidence: Evidence record to add
        """
        # Create node
        node = ProvenanceNode(
            evidence_id=evidence.evidence_id,
```

```
            evidence_type=evidence.evidence_type,
            source_method=evidence.source_method,
            timestamp=evidence.timestamp,
            parents=evidence.parent_evidence_ids.copy(),
        )

        # Add to nodes
        self.nodes[evidence.evidence_id] = node

        # Update parent-child relationships
        for parent_id in evidence.parent_evidence_ids:
            if parent_id in self.nodes:
                self.nodes[parent_id].children.append(evidence.evidence_id)

        # Update indices
        if evidence.source_method:
            self.by_method[evidence.source_method].append(evidence.evidence_id)
        self.by_type[evidence.evidence_type].append(evidence.evidence_id)
        if evidence.question_id:
            self.by_question[evidence.question_id].append(evidence.evidence_id)

    def get_ancestors(self, evidence_id: str) -> set[str]:
        """
        Get all ancestor evidence IDs (transitive parents).

        Args:
            evidence_id: Evidence ID to trace

        Returns:
            Set of ancestor evidence IDs
        """
        ancestors = set()
        visited = set()

        def traverse(eid: str) -> None:
            if eid in visited:
                return
            visited.add(eid)

            if eid not in self.nodes:
                return

            node = self.nodes[eid]
            for parent_id in node.parents:
                ancestors.add(parent_id)
                traverse(parent_id)

        traverse(evidence_id)
        return ancestors

    def get_descendants(self, evidence_id: str) -> set[str]:
        """
        Get all descendant evidence IDs (transitive children).

        Args:
            evidence_id: Evidence ID to trace

        Returns:
            Set of descendant evidence IDs
        """
        descendants = set()
        visited = set()

        def traverse(eid: str) -> None:
            if eid in visited:
                return
            visited.add(eid)
```

```
        if eid not in self.nodes:
            return

        node = self.nodes[eid]
        for child_id in node.children:
            descendants.add(child_id)
            traverse(child_id)

    traverse(evidence_id)
    return descendants

def get_lineage(self, evidence_id: str) -> dict[str, Any]:
    """
    Get complete lineage for evidence (ancestors + descendants).

    Args:
        evidence_id: Evidence ID to trace

    Returns:
        Dictionary with lineage information
    """
    return {
        "evidence_id": evidence_id,
        "ancestors": list(self.get_ancestors(evidence_id)),
        "descendants": list(self.get_descendants(evidence_id)),
        "ancestor_count": len(self.get_ancestors(evidence_id)),
        "descendant_count": len(self.get_descendants(evidence_id)),
    }

def export_dot(self) -> str:
    """
    Export DAG in GraphViz DOT format.

    Returns:
        DOT format string
    """
    lines = ["digraph ProvenanceDAG {"]
    lines.append("  rankdir=LR;")
    lines.append("  node [shape=box];")
    lines.append("")

    # Add nodes
    for eid, node in self.nodes.items():
        label = f"{node.evidence_type}\\n{eid[:8]}..."
        if node.source_method:
            label += f"\\n{node.source_method}"
        lines.append(f'  "{eid}" [label="{label}"];')

    lines.append("")

    # Add edges
    for eid, node in self.nodes.items():
        for child_id in node.children:
            lines.append(f'  "{eid}" -> "{child_id}";')

    lines.append("}")
    return "\n".join(lines)

def to_dict(self) -> dict[str, Any]:
    """Export DAG to dictionary."""
    return {
        "nodes": {eid: node.to_dict() for eid, node in self.nodes.items()},
        "stats": {
            "total_nodes": len(self.nodes),
            "by_method": {k: len(v) for k, v in self.by_method.items()},
            "by_type": {k: len(v) for k, v in self.by_type.items()},
            "by_question": {k: len(v) for k, v in self.by_question.items()},
        }
```

```
        }

class EvidenceRegistry:
    """
    Append-only evidence registry with hash indexing and provenance tracking.

    Features:
    - JSONL append-only storage for immutability
    - Content-addressable hash indexing
    - Provenance DAG for lineage tracking
    - Cryptographic verification
    - Fast queries by hash, type, method, question
    """

    def __init__(
        self,
        storage_path: Path | None = None,
        enable_dag: bool = True,
    ) -> None:
        """
        Initialize evidence registry.

        Args:
            storage_path: Path to JSONL storage file (default: evidence_registry.jsonl)
            enable_dag: Enable provenance DAG tracking
        """
        self.storage_path = storage_path or Path("evidence_registry.jsonl")
        self.enable_dag = enable_dag

        # Hash index: hash -> evidence record
        self.hash_index: dict[str, EvidenceRecord] = {}

        # Type index: type -> list of hashes
        self.type_index: dict[str, list[str]] = defaultdict(list)

        # Method index: method FQN -> list of hashes
        self.method_index: dict[str, list[str]] = defaultdict(list)

        # Question index: question ID -> list of hashes
        self.question_index: dict[str, list[str]] = defaultdict(list)

        # Provenance DAG
        self.dag = ProvenanceDAG() if enable_dag else None

        # Track the last entry in the ledger chain for hash chaining
        self.last_entry: EvidenceRecord | None = None

        # Load existing evidence
        self._load_from_storage()

        logger.info(
            f"EvidenceRegistry initialized with {len(self.hash_index)} records, "
            f"storage={self.storage_path}, dag={'enabled' if enable_dag else 'disabled'}"
        )

    def _load_from_storage(self) -> None:
        """
        Load evidence from JSONL storage with chain verification.

        Ensures:
        - Evidence is loaded in the order it was written
        - Chain linkage is validated during load
        - Index ordering is preserved
        """
        if not self.storage_path.exists():
            logger.info(f"No existing evidence storage found at {self.storage_path}")
            return
```

```
        loaded_count = 0
        loaded_records = []

        try:
            with open(self.storage_path, encoding='utf-8') as f:
                for line_num, line in enumerate(f, 1):
                    try:
                        data = json.loads(line.strip())
                        evidence = EvidenceRecord.from_dict(data)
                        loaded_records.append((line_num, evidence))
                        loaded_count += 1
                    except json.JSONDecodeError as e:
                        logger.warning(f"Failed to parse line {line_num}: {e}")
                    except Exception as e:
                        logger.warning(f"Failed to load evidence on line {line_num}: {e}")

            # Assert chain integrity during load
            self._assert_chain(loaded_records)

            # Index all loaded records in order
            for line_num, evidence in loaded_records:
                self._index_evidence(evidence, persist=False)

            logger.info(f"Loaded {loaded_count} evidence records from storage")

        except Exception as e:
            logger.error(f"Failed to load evidence storage: {e}")

    def _assert_chain(self, records: list[tuple[int, EvidenceRecord]]) -> None:
        """
        Assert that the chain of evidence records is valid.

        Validates:
        - First record has no previous_hash or previous_hash is None
        - Each subsequent record's previous_hash matches the prior record's entry_hash
        - Records are in the correct sequential order

        Args:
            records: List of (line_number, EvidenceRecord) tuples in load order

        Raises:
            ValueError: If chain validation fails
        """
        if not records:
            return

        previous_record = None

        for idx, (line_num, record) in enumerate(records):
            if idx == 0:
                # First record should have no previous_hash or None
                if record.previous_hash and record.previous_hash != '':
                    logger.warning(
                        f"Line {line_num}: First record has
previous_hash={record.previous_hash}, "
                        f"expected None or empty string. Chain may have been corrupted or
truncated."
                    )
            # Subsequent records should link to the previous record
            elif previous_record is not None:
                expected_previous_hash = previous_record.entry_hash
                actual_previous_hash = record.previous_hash

                if actual_previous_hash != expected_previous_hash:
                    raise ValueError(
                        f"Chain broken at line {line_num}: "
                        f"expected previous_hash={expected_previous_hash}, "
                        f"got previous_hash={actual_previous_hash}. "
```

```
                    f"Evidence ordering may be corrupted."
                )

            previous_record = record

    def _index_evidence(
        self,
        evidence: EvidenceRecord,
        persist: bool = True
    ) -> None:
        """
        Index evidence record in all indices.

        Args:
            evidence: Evidence to index
            persist: If True, append to JSONL storage
        """
        # Hash index
        self.hash_index[evidence.evidence_id] = evidence

        # Type index
        self.type_index[evidence.evidence_type].append(evidence.evidence_id)

        # Method index
        if evidence.source_method:
            self.method_index[evidence.source_method].append(evidence.evidence_id)

        # Question index
        if evidence.question_id:
            self.question_index[evidence.question_id].append(evidence.evidence_id)

        # DAG
        if self.enable_dag and self.dag:
            self.dag.add_evidence(evidence)

        # Update last entry for hash chaining
        self.last_entry = evidence

        # Persist to storage
        if persist:
            self._append_to_storage(evidence)

    def _append_to_storage(self, evidence: EvidenceRecord) -> None:
        """
        Append evidence to JSONL storage.

        Args:
            evidence: Evidence to append
        """
        try:
            # Ensure parent directory exists
            self.storage_path.parent.mkdir(parents=True, exist_ok=True)

            # Append to JSONL
            with open(self.storage_path, 'a', encoding='utf-8') as f:
                json_line = json.dumps(evidence.to_dict(), separators=(',', ':'))
                f.write(json_line + '\n')

        except Exception as e:
            logger.error(f"Failed to append evidence to storage: {e}")
            raise

    def record_evidence(
        self,
        evidence_type: str,
        payload: dict[str, Any],
        source_method: str | None = None,
        parent_evidence_ids: list[str] | None = None,
```

```
    question_id: str | None = None,
    document_id: str | None = None,
    execution_time_ms: float = 0.0,
    metadata: dict[str, Any] | None = None,
) -> str:
    """
    Record new evidence in registry.

    Args:
        evidence_type: Type of evidence
        payload: Evidence data
        source_method: FQN of method that produced evidence
        parent_evidence_ids: List of parent evidence IDs
        question_id: Question ID this evidence relates to
        document_id: Document ID this evidence relates to
        execution_time_ms: Execution time
        metadata: Additional metadata

    Returns:
        Evidence ID (hash)
    """
    # Determine previous_hash from last entry in the chain
    previous_hash = self.last_entry.entry_hash if self.last_entry else None

    # Normalize metadata and ensure recorded_at timestamp
    metadata_dict: dict[str, Any] = dict(metadata) if metadata else {}
    metadata_dict.setdefault(
        "recorded_at",
        datetime.now(timezone.utc).isoformat(),
    )

    # Create evidence record with deterministic hash-based ID
    evidence = EvidenceRecord.create(
        evidence_type=evidence_type,
        payload=payload,
        source_method=source_method,
        parent_evidence_ids=parent_evidence_ids,
        question_id=question_id,
        document_id=document_id,
        execution_time_ms=execution_time_ms,
        metadata=metadata_dict,
        previous_hash=previous_hash,
    )

    # Check for duplicate
    if evidence.evidence_id in self.hash_index:
        logger.debug(f"Evidence {evidence.evidence_id} already exists, skipping")
        return evidence.evidence_id

    # Index evidence
    self._index_evidence(evidence, persist=True)

    logger.debug(f"Recorded evidence {evidence.evidence_id} of type {evidence_type}")

    return evidence.evidence_id

def get_evidence(self, evidence_id: str) -> EvidenceRecord | None:
    """
    Retrieve evidence by ID.

    Args:
        evidence_id: Evidence hash

    Returns:
        EvidenceRecord or None
    """
    return self.hash_index.get(evidence_id)
```

```
    def query_by_type(self, evidence_type: str) -> list[EvidenceRecord]:
        """Query evidence by type."""
        evidence_ids = self.type_index.get(evidence_type, [])
        return [self.hash_index[eid] for eid in evidence_ids if eid in self.hash_index]

    def query_by_method(self, method_fqn: str) -> list[EvidenceRecord]:
        """Query evidence by source method."""
        evidence_ids = self.method_index.get(method_fqn, [])
        return [self.hash_index[eid] for eid in evidence_ids if eid in self.hash_index]

    def query_by_question(self, question_id: str) -> list[EvidenceRecord]:
        """Query evidence by question ID."""
        evidence_ids = self.question_index.get(question_id, [])
        return [self.hash_index[eid] for eid in evidence_ids if eid in self.hash_index]

    def verify_evidence(self, evidence_id: str, verify_chain: bool = True) -> bool:
        """
        Verify evidence integrity and optionally chain linkage.

        Args:
            evidence_id: Evidence hash
            verify_chain: If True, verify chain linkage with previous entry

        Returns:
            True if evidence is valid
        """
        evidence = self.get_evidence(evidence_id)
        if evidence is None:
            return False

        # Get previous record if chain verification is requested
        previous_record = None
        if verify_chain and evidence.previous_hash:
            # Find the record with entry_hash matching our previous_hash
            for record in self.hash_index.values():
                if record.entry_hash == evidence.previous_hash:
                    previous_record = record
                    break

        return evidence.verify_integrity(previous_record=previous_record)

    def verify_chain_integrity(self) -> tuple[bool, list[str]]:
        """
        Verify the integrity of the entire evidence chain.

        Returns:
            Tuple of (is_valid, list of errors)
        """
        errors = []

        # Build the chain by reading from storage in order
        if not self.storage_path.exists():
            return True, []  # Empty chain is valid

        try:
            previous_record = None
            with open(self.storage_path, encoding='utf-8') as f:
                for line_num, line in enumerate(f, 1):
                    try:
                        data = json.loads(line.strip())
                        evidence = EvidenceRecord.from_dict(data)

                        # Verify the record's integrity
                        if not evidence.verify_integrity(previous_record=previous_record):
                            if previous_record and evidence.previous_hash !=
previous_record.entry_hash:
                                errors.append(
                                    f"Line {line_num}: Chain broken - previous_hash
```

```
mismatch. "
                                f"Expected {previous_record.entry_hash}, got
{evidence.previous_hash}"
                            )
                        else:
                            errors.append(
                                f"Line {line_num}: Hash integrity check failed for
evidence {evidence.evidence_id}"
                            )

                        previous_record = evidence

                except json.JSONDecodeError as e:
                    errors.append(f"Line {line_num}: JSON parsing error - {e}")
                except Exception as e:
                    errors.append(f"Line {line_num}: Verification error - {e}")

            return len(errors) == 0, errors

        except Exception as e:
            return False, [f"Failed to verify chain: {e}"]

    def get_provenance(self, evidence_id: str) -> dict[str, Any] | None:
        """
        Get provenance information for evidence.

        Args:
            evidence_id: Evidence hash

        Returns:
            Provenance dictionary or None
        """
        if not self.enable_dag or self.dag is None:
            return None

        return self.dag.get_lineage(evidence_id)

    def export_provenance_dag(
        self,
        format: str = "dict",
        output_path: Path | None = None
    ) -> Any:
        """
        Export provenance DAG.

        Args:
            format: Export format ("dict", "dot", "json")
            output_path: Optional path to write output

        Returns:
            Exported DAG in requested format
        """
        if not self.enable_dag or self.dag is None:
            raise ValueError("DAG tracking is not enabled")

        if format == "dot":
            result = self.dag.export_dot()
        elif format == "dict":
            result = self.dag.to_dict()
        elif format == "json":
            result = json.dumps(self.dag.to_dict(), indent=2)
        else:
            raise ValueError(f"Unsupported format: {format}")

        # Write to file if path provided
        if output_path:
            output_path.parent.mkdir(parents=True, exist_ok=True)
            if isinstance(result, str):
```

```
                output_path.write_text(result, encoding='utf-8')
            else:
                output_path.write_text(json.dumps(result, indent=2), encoding='utf-8')
            logger.info(f"Exported provenance DAG to {output_path}")

        return result

    def get_statistics(self) -> dict[str, Any]:
        """
        Get registry statistics.

        Returns:
            Statistics dictionary
        """
        stats = {
            "total_evidence": len(self.hash_index),
            "by_type": {k: len(v) for k, v in self.type_index.items()},
            "by_method": {k: len(v) for k, v in self.method_index.items()},
            "by_question": {k: len(v) for k, v in self.question_index.items()},
            "storage_path": str(self.storage_path),
            "dag_enabled": self.enable_dag,
        }

        if self.enable_dag and self.dag:
            stats["dag_nodes"] = len(self.dag.nodes)

        return stats

    def stats(self) -> dict[str, int]:
        """Get simplified evidence registry statistics.

        Returns:
            Dict with counts for records, types, methods, and questions.
        """
        return {
            "records": len(self.hash_index),
            "types": len(self.type_index),
            "methods": len(self.method_index),
            "questions": len(self.question_index),
        }

# Global registry instance
_global_registry: EvidenceRegistry | None = None

def get_global_registry() -> EvidenceRegistry:
    """Get or create global evidence registry."""
    global _global_registry
    if _global_registry is None:
        _global_registry = EvidenceRegistry()
    return _global_registry

__all__ = [
    "EvidenceRecord",
    "ProvenanceNode",
    "ProvenanceDAG",
    "EvidenceRegistry",
    "get_global_registry",
]

===== FILE: src/saaaaaa/core/orchestrator/evidence_validator.py =====
from __future__ import annotations

import re
from typing import Any, Iterable


class EvidenceValidator:
    """Validate assembled evidence with configurable rules."""
```

```
    @staticmethod
    def validate(evidence: dict[str, Any], rules_object: dict[str, Any]) -> dict[str,
Any]:
        """
        Validates evidence against a rules object from a V2 contract.

        Args:
            evidence: The assembled evidence dictionary.
            rules_object: The validation object from the contract, containing
                    'rules' (a list) and 'na_policy' (a string).
        """
        validation_rules = rules_object.get("rules", [])
        na_policy = rules_object.get("na_policy", "abort_on_critical")
        errors: list[str] = []
        warnings: list[str] = []

        for rule in validation_rules:
            field = rule.get("field")
            value = EvidenceValidator._resolve(field, evidence)

            # --- New Rich Rule Logic ---
            if rule.get("must_contain"):
                must_contain = rule["must_contain"]
                required_elements = set(must_contain.get("elements", []))
                present_elements = set(value) if isinstance(value, list) else set()
                missing_elements = required_elements - present_elements
                if missing_elements:
                    errors.append(f"Field '{field}' is missing required elements: {',
'.join(sorted(missing_elements))}")

                required_count = must_contain.get("count")
                if required_count and
len(present_elements.intersection(required_elements)) < required_count:
                    errors.append(f"Field '{field}' did not meet the required count of
{required_count} for elements: {', '.join(sorted(required_elements))}")

            if rule.get("should_contain"):
                should_contain = rule["should_contain"]
                present_elements = set(value) if isinstance(value, list) else set()
                for requirement in should_contain:
                    elements_to_check = set(requirement.get("elements", []))
                    min_count = requirement.get("minimum", 1)
                    found_count = len(present_elements.intersection(elements_to_check))
                    if found_count < min_count:
                        warnings.append(f"Field '{field}' only has
{found_count}/{min_count} of recommended elements: {',
'.join(sorted(elements_to_check))}")

            # --- Original Simple Rule Logic ---
            missing = value is None
            if rule.get("required") and missing:
                errors.append(f"Missing required field '{field}'")
                continue
            if missing:
                continue

            if rule.get("type", "any") != "any" and not
EvidenceValidator._check_type(value, rule["type"]):
                errors.append(f"Field '{field}' has incorrect type (expected
{rule['type']})")
                continue

            if rule.get("min_length") is not None and EvidenceValidator._has_length(value)
 and len(value) < rule["min_length"]:
                errors.append(f"Field '{field}' length below min_length
{rule['min_length']}")
```

```
            if rule.get("pattern") and isinstance(value, str) and not
re.search(rule["pattern"], value):
                errors.append(f"Field '{field}' does not match pattern")


        valid = not errors
        if errors and na_policy == "abort_on_critical":
            raise ValueError(f"Evidence validation failed with critical errors: {';
'.join(errors)}")

        return {"valid": valid, "errors": errors, "warnings": warnings}

    @staticmethod
    def _resolve(path: str, evidence: dict[str, Any]) -> Any:
        if not path:
            return None
        parts = path.split(".")
        current: Any = evidence
        for part in parts:
            if isinstance(current, dict) and part in current:
                current = current[part]
            else:
                return None
        return current

    @staticmethod
    def _check_type(value: Any, expected: str) -> bool:
        mapping = {
            "array": (list, tuple),
            "integer": (int,),
            "float": (float, int),
            "string": (str,),
            "boolean": (bool,),
            "object": (dict,),
            "any": (object,),
        }
        return isinstance(value, mapping.get(expected, (object,)))

    @staticmethod
    def _has_length(value: Any) -> bool:
        return hasattr(value, "__len__")

    @staticmethod
    def _is_number(value: Any) -> bool:
        try:
            float(value)
            return not isinstance(value, bool)
        except (TypeError, ValueError):
            return False

===== FILE: src/saaaaaa/core/orchestrator/executor_config.py =====
from __future__ import annotations

from dataclasses import dataclass
from typing import Any


@dataclass
class ExecutorConfig:
    """
    Lightweight configuration for executors.

    This is intentionally minimal and only covers the parameters currently
    referenced by wiring/bootstrap code. Extend cautiously if new executor
    settings are required.
    """

    max_tokens: int | None = None
```

```
    temperature: float | None = None
    timeout_s: float | None = None
    retry: int | None = None
    seed: int | None = None
    extra: dict[str, Any] | None = None

    def __post_init__(self) -> None:
        # Basic type guards without altering semantics
        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")


__all__ = ["ExecutorConfig"]

===== FILE: src/saaaaaa/core/orchestrator/executors.py =====
"""
executors.py - Phase 2: Executor Orchestration for Policy Document Analysis

This module defines 30 executors (one per D{n}-Q{m} question) that orchestrate
methods from the core module to extract raw evidence from Colombian municipal
development plans (PDET/PDM documents).

Architecture:
- Each executor is independent and receives a canonical context package
- Methods execute in configured order; any failure causes executor failure
- Outputs are Python dicts/lists matching JSON contract specifications
- Executors are injected via MethodExecutor factory pattern

Usage:
    from factory import run_executor
    result = run_executor("D1-Q1", context_package)
"""

from __future__ import annotations

import sys
import logging
from typing import Dict, List, Any, Optional
from abc import ABC, abstractmethod
from dataclasses import dataclass

from saaaaaa.core.canonical_notation import CanonicalDimension, get_dimension_info
from saaaaaa.core.orchestrator.core import MethodExecutor
from saaaaaa.core.orchestrator.factory import build_processor
from saaaaaa.processing.policy_processor import CausalDimension

logger = logging.getLogger(__name__)

# Canonical question labels (only defined when verified in repo)
CANONICAL_QUESTION_LABELS = {
    "D3-Q2": "DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY",
    "D3-Q3": "DIM03_Q03_TRACEABILITY_BUDGET_ORG",
    "D3-Q4": "DIM03_Q04_TECHNICAL_FEASIBILITY",
    "D3-Q5": "DIM03_Q05_OUTPUT_OUTCOME_LINKAGE",
    "D4-Q1": "DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS",
    "D5-Q2": "DIM05_Q02_COMPOSITE_PROXY_VALIDITY",
}

# Epistemic taxonomy per method (focused on executors expanded in this iteration)
EPISTEMIC_TAGS = {
    ("FinancialAuditor", "_calculate_sufficiency"): ["statistical", "normative"],
    ("FinancialAuditor", "_match_program_to_node"): ["structural"],
    ("FinancialAuditor", "_match_goal_to_budget"): ["structural", "normative"],
    ("PDETMunicipalPlanAnalyzer", "_assess_financial_sustainability"): ["financial",
"normative"],
    ("PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility"): ["financial",
```

"statistical"],
    ("PDETMunicipalPlanAnalyzer", "_score_indicators"): ["normative", "semantic"],
    ("PDETMunicipalPlanAnalyzer", "_interpret_risk"): ["normative", "statistical"],
    ("PDETMunicipalPlanAnalyzer", "_extract_from_responsibility_tables"): ["structural"],
    ("PDETMunicipalPlanAnalyzer", "_consolidate_entities"): ["structural"],
    ("PDETMunicipalPlanAnalyzer", "_extract_entities_syntax"): ["semantic"],
    ("PDETMunicipalPlanAnalyzer", "_extract_entities_ner"): ["semantic"],
    ("PDETMunicipalPlanAnalyzer", "identify_responsible_entities"): ["semantic",
"structural"],
    ("PDETMunicipalPlanAnalyzer", "_score_responsibility_clarity"): ["normative"],
    ("PDETMunicipalPlanAnalyzer", "_refine_edge_probabilities"): ["statistical",
"causal"],
    ("PDETMunicipalPlanAnalyzer", "construct_causal_dag"): ["structural", "causal"],
    ("PDETMunicipalPlanAnalyzer", "estimate_causal_effects"): ["causal", "statistical"],
    ("PDETMunicipalPlanAnalyzer", "generate_counterfactuals"): ["causal"],
    ("PDETMunicipalPlanAnalyzer", "_identify_confounders"): ["causal", "consistency"],
    ("PDETMunicipalPlanAnalyzer", "_effect_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_scenario_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_get_spanish_stopwords"): ["semantic"],
    ("AdaptivePriorCalculator", "calculate_likelihood_adaptativo"): ["statistical",
"bayesian"],
    ("AdaptivePriorCalculator", "_adjust_domain_weights"): ["statistical"],
    ("BayesianMechanismInference", "_test_sufficiency"): ["statistical", "bayesian"],
    ("BayesianMechanismInference", "_test_necessity"): ["statistical", "bayesian"],
    ("BayesianMechanismInference", "_log_refactored_components"): ["implementation"],
    ("BayesianMechanismInference", "_infer_activity_sequence"): ["causal"],
    ("BayesianMechanismInference", "infer_mechanisms"): ["causal", "bayesian"],
    ("AdvancedDAGValidator", "calculate_acyclicity_pvalue"): ["statistical",
"consistency"],
    ("AdvancedDAGValidator", "_is_acyclic"): ["structural", "consistency"],
    ("AdvancedDAGValidator", "_calculate_bayesian_posterior"): ["statistical",
"bayesian"],
    ("AdvancedDAGValidator", "_calculate_confidence_interval"): ["statistical"],
    ("AdvancedDAGValidator", "_calculate_statistical_power"): ["statistical"],
    ("AdvancedDAGValidator", "_generate_subgraph"): ["structural"],
    ("AdvancedDAGValidator", "_get_node_validator"): ["implementation"],
    ("AdvancedDAGValidator", "_create_empty_result"): ["descriptive"],
    ("AdvancedDAGValidator", "_initialize_rng"): ["implementation"],
    ("AdvancedDAGValidator", "get_graph_stats"): ["structural"],
    ("AdvancedDAGValidator", "_calculate_node_importance"): ["structural"],
    ("AdvancedDAGValidator", "export_nodes"): ["structural", "descriptive"],
    ("AdvancedDAGValidator", "add_node"): ["structural"],
    ("AdvancedDAGValidator", "add_edge"): ["structural"],
    ("IndustrialGradeValidator", "execute_suite"): ["implementation", "normative"],
    ("IndustrialGradeValidator", "validate_connection_matrix"): ["consistency"],
    ("IndustrialGradeValidator", "run_performance_benchmarks"): ["implementation"],
    ("IndustrialGradeValidator", "_benchmark_operation"): ["implementation"],
    ("IndustrialGradeValidator", "validate_causal_categories"): ["consistency"],
    ("IndustrialGradeValidator", "_log_metric"): ["implementation"],
    ("PerformanceAnalyzer", "analyze_performance"): ["implementation", "normative"],
    ("PerformanceAnalyzer", "_calculate_loss_functions"): ["statistical"],
    ("HierarchicalGenerativeModel", "_calculate_ess"): ["statistical"],
    ("HierarchicalGenerativeModel", "_calculate_likelihood"): ["statistical"],
    ("HierarchicalGenerativeModel", "_calculate_r_hat"): ["statistical"],
    ("ReportingEngine", "generate_accountability_matrix"): ["normative", "structural"],
    ("ReportingEngine", "_calculate_quality_score"): ["normative", "statistical"],
    ("PolicyAnalysisEmbedder", "generate_pdq_report"): ["semantic", "descriptive"],
    ("PolicyAnalysisEmbedder", "compare_policy_interventions"): ["normative"],
    ("PolicyAnalysisEmbedder", "evaluate_policy_numerical_consistency"): ["consistency",
"statistical"],
    ("PolicyAnalysisEmbedder", "process_document"): ["semantic", "structural"],
    ("PolicyAnalysisEmbedder", "semantic_search"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_apply_mmr"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_generate_query_from_pdq"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_filter_by_pdq"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_extract_numerical_values"): ["statistical"],
    ("PolicyAnalysisEmbedder", "_compute_overall_confidence"): ["statistical",
"normative"],

```
    ("PolicyAnalysisEmbedder", "_embed_texts"): ["semantic"],
    ("SemanticAnalyzer", "_classify_policy_domain"): ["semantic"],
    ("SemanticAnalyzer", "_empty_semantic_cube"): ["descriptive"],
    ("SemanticAnalyzer", "_classify_cross_cutting_themes"): ["semantic"],
    ("SemanticAnalyzer", "_classify_value_chain_link"): ["semantic"],
    ("SemanticAnalyzer", "_vectorize_segments"): ["semantic"],
    ("SemanticAnalyzer", "_calculate_semantic_complexity"): ["semantic"],
    ("SemanticAnalyzer", "_process_segment"): ["semantic"],
    ("PDETMunicipalPlanAnalyzer", "_entity_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_quality_to_dict"): ["descriptive", "normative"],
    ("PDETMunicipalPlanAnalyzer", "_deduplicate_tables"): ["structural",
"implementation"],
    ("PDETMunicipalPlanAnalyzer", "_indicator_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_generate_recommendations"): ["normative"],
    ("PDETMunicipalPlanAnalyzer", "_simulate_intervention"): ["causal", "statistical"],
    ("PDETMunicipalPlanAnalyzer", "_identify_causal_nodes"): ["structural", "causal"],
    ("PDETMunicipalPlanAnalyzer", "_match_text_to_node"): ["semantic", "structural"],
    ("TeoriaCambio", "_validar_orden_causal"): ["causal", "consistency"],
    ("TeoriaCambio", "_generar_sugerencias_internas"): ["normative"],
    ("TeoriaCambio", "_extraer_categorias"): ["semantic"],
    ("BayesianMechanismInference", "_extract_observations"): ["semantic", "causal"],
    ("BayesianMechanismInference", "_generate_necessity_remediation"): ["normative",
"causal"],
    ("BayesianMechanismInference", "_quantify_uncertainty"): ["statistical", "bayesian"],
    ("CausalExtractor", "_build_type_hierarchy"): ["structural"],
    ("CausalExtractor", "_check_structural_violation"): ["structural", "consistency"],
    ("CausalExtractor", "_calculate_type_transition_prior"): ["statistical", "bayesian"],
    ("CausalExtractor", "_calculate_textual_proximity"): ["semantic"],
    ("CausalExtractor", "_calculate_language_specificity"): ["semantic"],
    ("CausalExtractor", "_calculate_composite_likelihood"): ["statistical", "semantic"],
    ("CausalExtractor", "_assess_financial_consistency"): ["financial", "consistency"],
    ("CausalExtractor", "_calculate_semantic_distance"): ["semantic"],
    ("CausalExtractor", "_extract_goals"): ["semantic"],
    ("CausalExtractor", "_parse_goal_context"): ["semantic"],
    ("CausalExtractor", "_classify_goal_type"): ["semantic"],
    ("TemporalLogicVerifier", "_parse_temporal_marker"): ["temporal", "consistency"],
    ("TemporalLogicVerifier", "_classify_temporal_type"): ["temporal", "consistency"],
    ("TemporalLogicVerifier", "_extract_resources"): ["structural"],
    ("TemporalLogicVerifier", "_should_precede"): ["temporal", "consistency"],
    ("AdaptivePriorCalculator", "generate_traceability_record"): ["structural",
"semantic"],
    ("PolicyAnalysisEmbedder", "generate_pdq_report"): ["semantic", "normative"],
    ("ReportingEngine", "generate_confidence_report"): ["normative", "descriptive"],
    ("PolicyTextProcessor", "segment_into_sentences"): ["semantic", "structural"],
    ("PolicyTextProcessor", "normalize_unicode"): ["implementation"],
    ("PolicyTextProcessor", "compile_pattern"): ["implementation"],
    ("PolicyTextProcessor", "extract_contextual_window"): ["semantic"],
    ("BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize"): ["causal",
"normative"],
    ("BayesianCounterfactualAuditor", "refutation_and_sanity_checks"): ["causal",
"consistency"],
    ("BayesianCounterfactualAuditor", "_evaluate_factual"): ["causal", "statistical"],
    ("BayesianCounterfactualAuditor", "_evaluate_counterfactual"): ["causal",
"statistical"],
    ("CausalExtractor", "_assess_financial_consistency"): ["financial", "consistency"],
    ("IndustrialPolicyProcessor", "_load_questionnaire"): ["descriptive",
"implementation"],
    ("IndustrialPolicyProcessor", "_compile_pattern_registry"): ["structural",
"semantic"],
    ("IndustrialPolicyProcessor", "_build_point_patterns"): ["semantic"],
    ("IndustrialPolicyProcessor", "_empty_result"): ["implementation"],
    ("IndustrialPolicyProcessor", "_compute_evidence_confidence"): ["statistical"],
    ("IndustrialPolicyProcessor", "_compute_avg_confidence"): ["statistical"],
    ("IndustrialPolicyProcessor", "_construct_evidence_bundle"): ["structural"],
    ("PDETMunicipalPlanAnalyzer", "generate_executive_report"): ["normative"],
    ("IndustrialPolicyProcessor", "export_results"): ["implementation"],
}
```

```
class BaseExecutor(ABC):
    """
    Base class for all executors with standardized execution template.
    All executors must implement execute() and return structured evidence.
    """

    def __init__(self, executor_id: str, config: Dict[str, Any], method_executor:
MethodExecutor):
        self.executor_id = executor_id
        self.config = config
        if not isinstance(method_executor, MethodExecutor):
            raise RuntimeError("A valid MethodExecutor instance is required for executor
injection.")
        self.method_executor = method_executor
        self.execution_log = []
        self._dimension_info = None  # Lazy load to avoid redundant fetches

    @property
    def dimension_info(self):
        """Lazy-loaded dimension information to avoid redundant metadata fetches."""
        if self._dimension_info is None:
            try:
                dim_key = self.executor_id.split("-")[0].replace("D", "D")
                self._dimension_info = get_dimension_info(dim_key)
            except (KeyError, ValueError, IndexError) as e:
                logger.warning(f"Failed to load dimension info for {self.executor_id}:
{e}")
                self._dimension_info = None
        return self._dimension_info

    def _validate_context(self, context: Dict[str, Any]) -> None:
        """
        Fail fast on malformed contexts.

        Raises:
            ValueError: If required context keys are missing
        """
        if not isinstance(context, dict):
            raise ValueError(f"Context must be a dict, got {type(context).__name__}")

        required = ["document_text"]
        missing = [k for k in required if k not in context]
        if missing:
            raise ValueError(f"Context missing required keys: {missing}")

    @abstractmethod
    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        """
        Execute configured methods and return raw evidence.

        Args:
            context: Canonical package with document, tables, metadata

        Returns:
            Dict with raw_evidence, metadata, execution_metrics

        Raises:
            ExecutorFailure: If any method fails
        """
        pass

    def _log_method_execution(self, class_name: str, method_name: str,
                              success: bool, result: Any = None, error: str = None):
        """Track method execution for debugging and traceability."""
        self.execution_log.append({
            "class": class_name,
            "method": method_name,
```

```
            "success": success,
            "result_type": type(result).__name__ if result else None,
            "error": error
        })

    def _execute_method(self, class_name: str, method_name: str,
                        context: Dict[str, Any], **kwargs) -> Any:
        """
        Execute a single method with error handling.

        Raises:
            ExecutorFailure: If method execution fails
        """
        try:
            # Method injection happens via factory - placeholder for actual execution
            method = self._get_method(class_name, method_name)
            result = method(context, **kwargs)
            self._log_method_execution(class_name, method_name, True, result)
            return result
        except Exception as e:
            self._log_method_execution(class_name, method_name, False, error=str(e))
            raise ExecutorFailure(
                f"Executor {self.executor_id} failed: {class_name}.{method_name} -
{str(e)}"
            ) from e  # Preserve exception chain for debugging

    def _get_method(self, class_name: str, method_name: str):
        """Retrieve method using MethodExecutor to enforce routed execution."""
        if not isinstance(self.method_executor, MethodExecutor):
            raise RuntimeError(f"Invalid method executor provided:
{type(self.method_executor).__name__}")

        def _wrapped(context: Dict[str, Any], **kwargs: Any) -> Any:
            payload: Dict[str, Any] = {}
            if context:
                payload.update(context)
            if kwargs:
                payload.update(kwargs)
            return self.method_executor.execute(
                class_name=class_name,
                method_name=method_name,
                **payload,
            )

        return _wrapped


@dataclass
class ExecutorResult:
    """
    Standardized result container for executor execution.
    Ensures type safety and verifiable structure.
    """
    executor_id: str
    success: bool
    data: Optional[Dict[str, Any]]
    error: Optional[str]
    execution_time_ms: int
    memory_usage_mb: float


class ExecutorFailure(Exception):
    """Raised when any method in an executor fails."""
    pass


# ============================================================================
# DIMENSION 1: DIAGNOSTICS & INPUTS
```

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

class D1_Q1_QuantitativeBaselineExtractor(BaseExecutor):
    """
    Extracts numeric data, reference years, and official sources as baseline.

    Methods (from D1-Q1):
    - TextMiningEngine.diagnose_critical_links
    - TextMiningEngine._analyze_link_text
    - IndustrialPolicyProcessor.process
    - IndustrialPolicyProcessor._match_patterns_in_sentences
    - IndustrialPolicyProcessor._extract_point_evidence
    - CausalExtractor._extract_goals
    - CausalExtractor._parse_goal_context
    - FinancialAuditor._parse_amount
    - PDETMunicipalPlanAnalyzer._extract_financial_amounts
    - PDETMunicipalPlanAnalyzer._extract_from_budget_table
    - PolicyContradictionDetector._extract_quantitative_claims
    - PolicyContradictionDetector._parse_number
    - PolicyContradictionDetector._statistical_significance_test
    - BayesianNumericalAnalyzer.evaluate_policy_metric
    - BayesianNumericalAnalyzer.compare_policies
    - SemanticProcessor.chunk_text
    - SemanticProcessor.embed_single
    """

    def execute(self, context: Dict[str, Any] | None = None, **kwargs: Any) -> Dict[str,
Any]:
        if context is None:
            context = dict(kwargs)
        raw_evidence: Dict[str, Any] = {}

        # The new implementation requires manual instantiation of some components
        # because the dependency injection logic was part of the old MethodExecutor.
        # This will be revisited when refactoring the executor base class.
        ontology = self.method_executor.shared_instances.get("MunicipalOntology")

        # Step 0: Initial processing of the document text
        raw_text = context.get("raw_text", "")
        sentences = self._execute_method("PolicyTextProcessor", "segment_into_sentences",
context, text=raw_text)

        # Step 1: Semantic and Performance Analysis (Prerequisites for TextMiningEngine)
        semantic_cube = self._execute_method("SemanticAnalyzer", "extract_semantic_cube",
context, document_segments=sentences)
        performance_analysis = self._execute_method("PerformanceAnalyzer",
"analyze_performance", context, semantic_cube=semantic_cube)

        # Step 2: Identify critical data-bearing sections
        critical_links = self._execute_method(
            "TextMiningEngine", "diagnose_critical_links", context,
            semantic_cube=semantic_cube,
            performance_analysis=performance_analysis
        )

        # The output of diagnose_critical_links is complex. Let's assume it contains
segments for _analyze_link_text
        link_analysis_segments = critical_links.get("critical_links",
{}).get(next(iter(critical_links.get("critical_links", {}))), None),
{}).get("text_analysis", {}).get("keywords", [])
        link_analysis = self._execute_method(
            "TextMiningEngine", "_analyze_link_text", context,
            segments=[{"text": s} for s in link_analysis_segments] # _analyze_link_text
expects list of dicts
        )

        # Step 3: Extract structured quantitative claims from the whole document
        processed_sections = self._execute_method(
```

```
            "IndustrialPolicyProcessor", "process", context,
            raw_text=raw_text
        )
        # We need compiled patterns. This is a challenge. Let's assume the processor can
get them.
        compiled_patterns = self._execute_method("IndustrialPolicyProcessor",
"_compile_pattern_registry", context)

        pattern_matches, _ = self._execute_method(
            "IndustrialPolicyProcessor", "_match_patterns_in_sentences", context,
            compiled_patterns=compiled_patterns.get(CausalDimension.D1_INSUMOS,
{}).get('diagnostico_cuantitativo', []),
            relevant_sentences=sentences
        )

        point_evidence_list = []
        for point_code in self._execute_method("IndustrialPolicyProcessor",
"_build_point_patterns", context):
            point_evidence = self._execute_method(
                "IndustrialPolicyProcessor", "_extract_point_evidence", context,
                text=raw_text,
                sentences=sentences,
                point_code=point_code
            )
            point_evidence_list.append(point_evidence)

        # Step 4: Parse numerical amounts and baseline data
        all_text = " ".join(pattern_matches)
        parsed_amounts = self._execute_method( "FinancialAuditor", "_parse_amount",
context, value=all_text)

        financial_amounts = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_financial_amounts", context,
            text=raw_text, tables=context.get("tables", [])
        )

        # This method needs a dataframe, which is not available in the context.
        # I will skip this call for now.
        # budget_table_data = self._execute_method(
        #     "PDETMunicipalPlanAnalyzer", "_extract_from_budget_table", context
        # )
        budget_table_data = None

        # Step 5: Extract temporal context (reference years)
        goals = self._execute_method( "CausalExtractor", "_extract_goals", context,
text=raw_text)

        goal_contexts = []
        if isinstance(goals, list):
            for goal in goals:
                goal_id = goal.id
                goal_context_str = goal.text
                if goal_id and goal_context_str:
                    res = self._execute_method(
                        "CausalExtractor", "_parse_goal_context", context,
                        goal_id=goal_id,
                        context=goal_context_str
                    )
                    if res:
                        goal_contexts.append(res)

        # Step 6: Validate quantitative claims
        quant_claims = self._execute_method( "PolicyContradictionDetector",
"_extract_quantitative_claims", context, text=raw_text)

        parsed_numbers = []
        if isinstance(quant_claims, list):
            for claim in quant_claims:
```

```
            res = self._execute_method(
                "PolicyContradictionDetector", "_parse_number", context,
                text=claim.get("raw_text")
            )
            if res is not None:
                parsed_numbers.append(res)

        significance_test = None
        if len(parsed_numbers) >= 2:
            # The method expects claims, not just numbers. Let's create dummy claims.
            claim_a = {'value': parsed_numbers[0]}
            claim_b = {'value': parsed_numbers[1]}
            significance_test = self._execute_method(
                "PolicyContradictionDetector", "_statistical_significance_test", context,
                claim_a=claim_a,
                claim_b=claim_b
            )

        # Step 7: Evaluate baseline quality and compare
        metric_evaluation = self._execute_method(
            "BayesianNumericalAnalyzer", "evaluate_policy_metric", context,
            observed_values=parsed_numbers
        )

        policy_comparison = None
        if metric_evaluation and "posterior_samples" in metric_evaluation:
             policy_comparison = self._execute_method(
                "BayesianNumericalAnalyzer", "compare_policies", context,
                policy_a_values=[s['coherence'] for s in
metric_evaluation.get("posterior_samples", [])],
                policy_b_values=context.get("baseline_samples", []) # Assuming baseline
samples exist
            )

        # Step 8: Semantic validation of sources
        text_chunks = self._execute_method( "SemanticProcessor", "chunk_text", context,
text=raw_text, preserve_structure=True)

        embeddings = []
        if isinstance(text_chunks, list):
            texts_to_embed = [chunk['content'] for chunk in text_chunks if 'content' in
chunk]
            embeddings = self._execute_method( "SemanticProcessor", "_embed_batch",
context, texts=texts_to_embed)

        # Assemble raw evidence
        raw_evidence = {
            "numeric_data": parsed_numbers,
            "reference_years": [gc.year for gc in goal_contexts if gc and hasattr(gc,
'year')],
            "official_sources": point_evidence_list,
            "financial_baseline": financial_amounts,
            "budget_tables": budget_table_data,
            "significance_results": significance_test,
            "metric_evaluation": metric_evaluation,
            "policy_comparison": policy_comparison,
            "goal_contexts": [res.text if res else None for res in goal_contexts],
            "quantitative_claims": quant_claims,
            "processed_sections": processed_sections,
            "pattern_matches": pattern_matches,
            "link_analysis": link_analysis,
            "source_embeddings": embeddings
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
```

```
            "methods_executed": [log["method"] for log in self.execution_log],
            "total_numeric_claims": len(parsed_numbers or []),
            "sources_identified": len(point_evidence_list)
        },
        "execution_metrics": {
            "methods_count": len(self.execution_log),
            "all_succeeded": all(log["success"] for log in self.execution_log)
        }
    }


class D1_Q2_ProblemDimensioningAnalyzer(BaseExecutor):
    """
    Quantifies problem magnitude, gaps, and identifies data limitations.

    Methods (from D1-Q2):
    - OperationalizationAuditor._audit_direct_evidence
    - OperationalizationAuditor._audit_systemic_risk
    - FinancialAuditor._detect_allocation_gaps
    - BayesianMechanismInference._detect_gaps
    - PDETMunicipalPlanAnalyzer._generate_optimal_remediations
    - PDETMunicipalPlanAnalyzer._simulate_intervention
    - BayesianCounterfactualAuditor.counterfactual_query
    - BayesianCounterfactualAuditor._test_effect_stability
    - PolicyContradictionDetector._detect_numerical_inconsistencies
    - PolicyContradictionDetector._calculate_numerical_divergence
    - BayesianConfidenceCalculator.calculate_posterior
    - PerformanceAnalyzer.analyze_performance
    """

    def execute(self, context: Dict[str, Any] | None = None, **kwargs: Any) -> Dict[str,
Any]:
        if context is None:
            context = dict(kwargs)
        raw_evidence: Dict[str, Any] = {}

        # Step 1: Audit evidence completeness
        direct_evidence_audit = self._execute_method(
            "OperationalizationAuditor", "_audit_direct_evidence", context
        )
        systemic_risk_audit = self._execute_method(
            "OperationalizationAuditor", "_audit_systemic_risk", context
        )

        # Step 2: Detect gaps in resource allocation and mechanisms
        allocation_gaps = self._execute_method(
            "FinancialAuditor", "_detect_allocation_gaps", context
        )
        mechanism_gaps = self._execute_method(
            "BayesianMechanismInference", "_detect_gaps", context
        )

        # Step 3: Generate optimal remediations and simulate interventions
        remediations = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_generate_optimal_remediations", context,
            gaps=allocation_gaps
        )
        simulation_results = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_simulate_intervention", context,
            remediations=remediations
        )

        # Step 4: Counterfactual analysis for problem dimensioning
        counterfactual = self._execute_method(
            "BayesianCounterfactualAuditor", "counterfactual_query", context
        )
        effect_stability = self._execute_method(
            "BayesianCounterfactualAuditor", "_test_effect_stability", context,
```

```
            counterfactual=counterfactual
        )

        # Step 5: Detect numerical inconsistencies
        numerical_inconsistencies = self._execute_method(
            "PolicyContradictionDetector", "_detect_numerical_inconsistencies", context
        )
        divergence_calc = self._execute_method(
            "PolicyContradictionDetector", "_calculate_numerical_divergence", context,
            inconsistencies=numerical_inconsistencies
        )

        # Step 6: Calculate confidence and analyze performance
        posterior_confidence = self._execute_method(
            "BayesianConfidenceCalculator", "calculate_posterior", context,
            evidence=direct_evidence_audit
        )
        performance_analysis = self._execute_method(
            "PerformanceAnalyzer", "analyze_performance", context
        )

        raw_evidence = {
            "magnitude_indicators": {
                "allocation_gaps": allocation_gaps,
                "mechanism_gaps": mechanism_gaps,
                "numerical_inconsistencies": numerical_inconsistencies
            },
            "deficit_quantification": divergence_calc,
            "counterfactual_analysis": counterfactual,
            "effect_stability": effect_stability,
            "data_limitations": {
                "evidence_gaps": direct_evidence_audit.get("gaps", []),
                "systemic_risks": systemic_risk_audit
            },
            "simulation_results": simulation_results,
            "confidence_scores": posterior_confidence,
            "performance_metrics": performance_analysis
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "gaps_identified": len(allocation_gaps or []) + len(mechanism_gaps or []),
                "inconsistencies_found": len(numerical_inconsistencies or [])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D1_Q3_BudgetAllocationTracer(BaseExecutor):
    """
    Traces monetary resources assigned to programs in Investment Plan (PPI).

    Methods (from D1-Q3):
    - FinancialAuditor.trace_financial_allocation
    - FinancialAuditor._process_financial_table
    - FinancialAuditor._match_program_to_node
    - FinancialAuditor._match_goal_to_budget
    - FinancialAuditor._perform_counterfactual_budget_check
    - FinancialAuditor._calculate_sufficiency
    - PDETMunicipalPlanAnalyzer.analyze_financial_feasibility
    - PDETMunicipalPlanAnalyzer._extract_budget_for_pillar
    - PDETMunicipalPlanAnalyzer._identify_funding_source
```

```
- PDETMunicipalPlanAnalyzer._classify_tables
- PDETMunicipalPlanAnalyzer._analyze_funding_sources
- PDETMunicipalPlanAnalyzer._score_financial_component
- BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
"""

def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
    raw_evidence = {}

    # Step 1: Trace complete financial allocation chain
    allocation_trace = self._execute_method(
        "FinancialAuditor", "trace_financial_allocation", context
    )
    processed_tables = self._execute_method(
        "FinancialAuditor", "_process_financial_table", context
    )

    # Step 2: Match programs to budget nodes
    program_matches = self._execute_method(
        "FinancialAuditor", "_match_program_to_node", context,
        tables=processed_tables
    )
    goal_budget_matches = self._execute_method(
        "FinancialAuditor", "_match_goal_to_budget", context,
        programs=program_matches
    )

    # Step 3: Counterfactual checks and sufficiency calculation
    counterfactual_check = self._execute_method(
        "FinancialAuditor", "_perform_counterfactual_budget_check", context,
        matches=goal_budget_matches
    )
    sufficiency_calc = self._execute_method(
        "FinancialAuditor", "_calculate_sufficiency", context,
        allocation=allocation_trace
    )

    # Step 4: Analyze financial feasibility
    feasibility_analysis = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility", context
    )
    pillar_budgets = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_extract_budget_for_pillar", context
    )
    funding_sources = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_identify_funding_source", context
    )

    # Step 5: Classify and analyze tables
    table_classification = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_classify_tables", context,
        tables=processed_tables
    )
    funding_analysis = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_analyze_funding_sources", context,
        sources=funding_sources
    )
    financial_score = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_score_financial_component", context,
        analysis=funding_analysis
    )

    # Step 6: Aggregate risk and prioritize
    risk_aggregation = self._execute_method(
        "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
        sufficiency=sufficiency_calc,
        counterfactual=counterfactual_check
    )
```

```
        raw_evidence = {
            "budget_allocations": allocation_trace,
            "program_mappings": program_matches,
            "goal_budget_links": goal_budget_matches,
            "counterfactual_budget_check": counterfactual_check,
            "sufficiency_analysis": sufficiency_calc,
            "pillar_budgets": pillar_budgets,
            "funding_sources": funding_sources,
            "financial_feasibility": feasibility_analysis,
            "financial_score": financial_score,
            "table_classification": table_classification,
            "funding_analysis": funding_analysis,
            "risk_priorities": risk_aggregation
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "programs_traced": len(program_matches or []),
                "funding_sources_identified": len(funding_sources or [])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D1_Q4_InstitutionalCapacityIdentifier(BaseExecutor):
    """
    Identifies installed capacity (entities, staff, equipment) and limitations.

    Methods (from D1-Q4):
    - PDETMunicipalPlanAnalyzer.identify_responsible_entities
    - PDETMunicipalPlanAnalyzer._extract_entities_ner
    - PDETMunicipalPlanAnalyzer._extract_entities_syntax
    - PDETMunicipalPlanAnalyzer._classify_entity_type
    - PDETMunicipalPlanAnalyzer._score_entity_specificity
    - PDETMunicipalPlanAnalyzer._consolidate_entities
    - MechanismPartExtractor.extract_entity_activity
    - MechanismPartExtractor._normalize_entity
    - MechanismPartExtractor._validate_entity_activity
    - MechanismPartExtractor._calculate_ea_confidence
    - OperationalizationAuditor.audit_evidence_traceability
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Identify all responsible entities
        entities_identified = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "identify_responsible_entities", context
        )

        # Step 2: Extract entities using NER and syntax
        ner_entities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_entities_ner", context
        )
        syntax_entities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_entities_syntax", context
        )

        # Step 3: Classify and score entities
        entity_types = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_classify_entity_type", context,
```

```
            entities=ner_entities + syntax_entities
        )
        specificity_scores = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_entity_specificity", context,
            entities=entity_types
        )
        consolidated = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_consolidate_entities", context,
            entities=entity_types
        )

        # Step 4: Extract entity-activity relationships
        entity_activities = self._execute_method(
            "MechanismPartExtractor", "extract_entity_activity", context,
            entities=consolidated
        )
        normalized = self._execute_method(
            "MechanismPartExtractor", "_normalize_entity", context,
            activities=entity_activities
        )
        validated = self._execute_method(
            "MechanismPartExtractor", "_validate_entity_activity", context,
            normalized=normalized
        )
        ea_confidence = self._execute_method(
            "MechanismPartExtractor", "_calculate_ea_confidence", context,
            validated=validated
        )

        # Step 5: Audit evidence traceability
        traceability_audit = self._execute_method(
            "OperationalizationAuditor", "audit_evidence_traceability", context,
            entity_activities=validated
        )

        raw_evidence = {
            "entities_identified": consolidated,
            "entity_types": entity_types,
            "specificity_scores": specificity_scores,
            "entity_activities": validated,
            "activity_confidence": ea_confidence,
            "capacity_indicators": {
                "staff_mentions": [e for e in consolidated if e.get("type") == "staff"],
                "equipment_mentions": [e for e in consolidated if e.get("type") ==
"equipment"],
                "organizational_units": [e for e in consolidated if e.get("type") ==
"organization"]
            },
            "limitations_identified": (traceability_audit or {}).get("gaps", []),
            "traceability_audit": traceability_audit
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "entities_count": len(consolidated or []),
                "activities_extracted": len(validated or [])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D1_Q5_ScopeJustificationValidator(BaseExecutor):
```

```
"""
Validates scope justification via legal framework and constraint recognition.

Methods (from D1-Q5):
- TemporalLogicVerifier._check_deadline_constraints
- TemporalLogicVerifier.verify_temporal_consistency
- CausalInferenceSetup.identify_failure_points
- CausalExtractor._assess_temporal_coherence
- TextMiningEngine._analyze_link_text
- IndustrialPolicyProcessor._analyze_causal_dimensions
- IndustrialPolicyProcessor._extract_metadata
"""

def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
    raw_evidence = {}

    # Step 1: Verify temporal constraints
    deadline_constraints = self._execute_method(
        "TemporalLogicVerifier", "_check_deadline_constraints", context
    )
    temporal_consistency = self._execute_method(
        "TemporalLogicVerifier", "verify_temporal_consistency", context
    )

    # Step 2: Identify failure points in scope
    failure_points = self._execute_method(
        "CausalInferenceSetup", "identify_failure_points", context
    )

    # Step 3: Assess temporal coherence
    temporal_coherence = self._execute_method(
        "CausalExtractor", "_assess_temporal_coherence", context
    )

    # Step 4: Analyze link text for justifications
    link_analysis = self._execute_method(
        "TextMiningEngine", "_analyze_link_text", context
    )

    # Step 5: Analyze causal dimensions and extract metadata
    causal_dimensions = self._execute_method(
        "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
    )
    metadata_extracted = self._execute_method(
        "IndustrialPolicyProcessor", "_extract_metadata", context,
        dimensions=causal_dimensions
    )

    raw_evidence = {
        "legal_framework_citations": metadata_extracted.get("legal_refs", []),
        "temporal_constraints": {
            "deadline_checks": deadline_constraints,
            "consistency": temporal_consistency,
            "coherence": temporal_coherence
        },
        "budgetary_constraints": metadata_extracted.get("budget_limits", []),
        "competence_constraints": metadata_extracted.get("competence_refs", []),
        "failure_points": failure_points,
        "scope_justifications": (link_analysis or {}).get("justifications", []),
        "causal_dimensions": causal_dimensions
    }

    return {
        "executor_id": self.executor_id,
        "raw_evidence": raw_evidence,
        "metadata": {
            "methods_executed": [log["method"] for log in self.execution_log],
            "constraints_identified": len(deadline_constraints or []),
```

```
            "legal_citations": len(metadata_extracted.get("legal_refs", []))
        },
        "execution_metrics": {
            "methods_count": len(self.execution_log),
            "all_succeeded": all(log["success"] for log in self.execution_log)
        }
    }


# ============================================================================
# DIMENSION 2: ACTIVITY DESIGN
# ============================================================================

class D2_Q1_StructuredPlanningValidator(BaseExecutor):
    """
    Validates structured format of activities (table/matrix with required columns).

    Methods (from D2-Q1):
    - PDFProcessor.extract_tables
    - FinancialAuditor._process_financial_table
    - PDETMunicipalPlanAnalyzer._deduplicate_tables
    - PDETMunicipalPlanAnalyzer._classify_tables
    - PDETMunicipalPlanAnalyzer._is_likely_header
    - PDETMunicipalPlanAnalyzer._clean_dataframe
    - ReportingEngine.generate_accountability_matrix
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Extract all tables
        extracted_tables = self._execute_method(
            "PDFProcessor", "extract_tables", context
        )

        # Step 2: Process financial tables
        processed_tables = self._execute_method(
            "FinancialAuditor", "_process_financial_table", context,
            tables=extracted_tables
        )

        # Step 3: Deduplicate and classify tables
        deduplicated = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_deduplicate_tables", context,
            tables=processed_tables
        )
        classified = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_classify_tables", context,
            tables=deduplicated
        )

        # Step 4: Identify headers and clean dataframes
        header_checks = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_is_likely_header", context,
            tables=classified
        )
        cleaned = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_clean_dataframe", context,
            tables=classified
        )

        # Step 5: Generate accountability matrix
        accountability_matrix = self._execute_method(
            "ReportingEngine", "generate_accountability_matrix", context,
            tables=cleaned
        )

        raw_evidence = {
```

```
            "tables_extracted": len(extracted_tables),
            "activity_tables": [t for t in classified if t.get("type") == "activity"],
            "matrix_structure": accountability_matrix,
            "required_columns_present": {
                "responsible_entity": any("responsible" in str(t.get("columns",
[])).lower()
                                          for t in cleaned),
                "deliverable": any("deliverable" in str(t.get("columns", [])).lower()
                                   for t in cleaned),
                "timeline": any("timeline" in str(t.get("columns", [])).lower()
                                for t in cleaned),
                "cost": any("cost" in str(t.get("columns", [])).lower()
                            for t in cleaned)
            },
            "table_quality": {
                "clean_tables": len(cleaned),
                "with_headers": sum(1 for h in header_checks if h)
            }
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "total_tables": len(extracted_tables),
                "activity_tables": len([t for t in classified if t.get("type") ==
"activity"])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D2_Q2_InterventionLogicInferencer(BaseExecutor):
    """
    Infers intervention logic: instrument (how), target (who), causality (why).

    Methods (from D2-Q2):
    - BayesianMechanismInference.infer_mechanisms
    - BayesianMechanismInference._infer_single_mechanism
    - BayesianMechanismInference._infer_mechanism_type
    - BayesianMechanismInference._test_sufficiency
    - BayesianMechanismInference._test_necessity
    - CausalExtractor.extract_causal_hierarchy
    - TeoriaCambio.construir_grafo_causal
    - TeoriaCambio._es_conexion_valida
    - PDETMunicipalPlanAnalyzer.construct_causal_dag
    - BeachEvidentialTest.classify_test
    - IndustrialPolicyProcessor._analyze_causal_dimensions
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Infer mechanisms
        mechanisms = self._execute_method(
            "BayesianMechanismInference", "infer_mechanisms", context
        )
        single_mechanisms = []
        for mech in mechanisms:
            single = self._execute_method(
                "BayesianMechanismInference", "_infer_single_mechanism", context,
                mechanism=mech
            )
            single_mechanisms.append(single)
```

```
mechanism_types = self._execute_method(
    "BayesianMechanismInference", "_infer_mechanism_type", context,
    mechanisms=single_mechanisms
)

# Step 2: Test sufficiency and necessity
sufficiency_tests = self._execute_method(
    "BayesianMechanismInference", "_test_sufficiency", context,
    mechanisms=single_mechanisms
)
necessity_tests = self._execute_method(
    "BayesianMechanismInference", "_test_necessity", context,
    mechanisms=single_mechanisms
)

# Step 3: Extract causal hierarchy
causal_hierarchy = self._execute_method(
    "CausalExtractor", "extract_causal_hierarchy", context
)

# Step 4: Build causal graph
causal_graph = self._execute_method(
    "TeoriaCambio", "construir_grafo_causal", context,
    hierarchy=causal_hierarchy
)
connection_validation = self._execute_method(
    "TeoriaCambio", "_es_conexion_valida", context,
    graph=causal_graph
)

# Step 5: Construct DAG
causal_dag = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "construct_causal_dag", context,
    graph=causal_graph
)

# Step 6: Classify evidential tests
evidential_tests = self._execute_method(
    "BeachEvidentialTest", "classify_test", context,
    mechanisms=single_mechanisms
)

# Step 7: Analyze causal dimensions
causal_dimensions = self._execute_method(
    "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
)

raw_evidence = {
    "intervention_instruments": [m.get("instrument") for m in single_mechanisms],
    "target_populations": [m.get("target") for m in single_mechanisms],
    "causal_logic": {
        "mechanisms": single_mechanisms,
        "mechanism_types": mechanism_types,
        "sufficiency": sufficiency_tests,
        "necessity": necessity_tests
    },
    "causal_hierarchy": causal_hierarchy,
    "causal_graph": causal_graph,
    "causal_dag": causal_dag,
    "evidential_strength": evidential_tests,
    "connection_validation": connection_validation,
    "dimensions": causal_dimensions
}

return {
    "executor_id": self.executor_id,
    "raw_evidence": raw_evidence,
```

```
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "mechanisms_identified": len(single_mechanisms or []),
                "instruments_found": len([m for m in single_mechanisms if
m.get("instrument")]),
                "connections_valid": bool(connection_validation)
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D2_Q3_RootCauseLinkageAnalyzer(BaseExecutor):
    """
    Analyzes linkage between activities and root causes/structural determinants.

    Methods (from D2-Q3):
    - CausalExtractor._extract_causal_links
    - CausalExtractor._calculate_composite_likelihood
    - CausalExtractor._initialize_prior
    - CausalExtractor._calculate_type_transition_prior
    - PDETMunicipalPlanAnalyzer._identify_causal_edges
    - PDETMunicipalPlanAnalyzer._refine_edge_probabilities
    - BayesianCounterfactualAuditor.construct_scm
    - BayesianCounterfactualAuditor._create_default_equations
    - SemanticAnalyzer.extract_semantic_cube
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Extract causal links
        causal_links = self._execute_method(
            "CausalExtractor", "_extract_causal_links", context
        )

        # Step 2: Calculate likelihoods
        composite_likelihood = self._execute_method(
            "CausalExtractor", "_calculate_composite_likelihood", context,
            links=causal_links
        )
        prior_init = self._execute_method(
            "CausalExtractor", "_initialize_prior", context
        )
        type_transition_prior = self._execute_method(
            "CausalExtractor", "_calculate_type_transition_prior", context,
            links=causal_links
        )

        # Step 3: Identify and refine causal edges
        causal_edges = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_identify_causal_edges", context,
            links=causal_links
        )
        refined_probabilities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_refine_edge_probabilities", context,
            edges=causal_edges
        )

        # Step 4: Construct structural causal model
        scm = self._execute_method(
            "BayesianCounterfactualAuditor", "construct_scm", context,
            edges=refined_probabilities
        )
        default_equations = self._execute_method(
            "BayesianCounterfactualAuditor", "_create_default_equations", context,
```

```
            scm=scm
        )

        # Step 5: Extract semantic cube
        semantic_cube = self._execute_method(
            "SemanticAnalyzer", "extract_semantic_cube", context
        )

        raw_evidence = {
            "root_causes_identified": [link.get("root_cause") for link in (causal_links or
 [])],
            "activity_linkages": causal_links,
            "link_probabilities": refined_probabilities,
            "composite_likelihood": composite_likelihood,
            "prior_initialization": prior_init,
            "type_transition_prior": type_transition_prior,
            "structural_model": scm,
            "model_equations": default_equations,
            "semantic_relationships": semantic_cube,
            "determinants_addressed": [link for link in (causal_links or []) if
link.get("addresses_determinant")]
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "causal_links_found": len(causal_links or []),
                "root_causes_count": len(set(link.get("root_cause") for link in
(causal_links or [])))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }
```