```
        # Weakest link principle
        min_score = min(method_scores.values())

        logger.info(
            "chain_quality_computed",
            extra={
                "min_score": min_score,
                "num_methods": len(method_scores)
            }
        )

        return min_score

===== FILE: src/saaaaaa/core/calibration/choquet_aggregator.py =====
"""
Choquet 2-Additive Aggregation.

This implements the final score computation:
    Cal(I) = Σ a_ℓ·x_ℓ + Σ a_ℓk·min(x_ℓ, x_k)

Where:
- First sum: linear terms (weighted sum of layer scores)
- Second sum: interaction terms (synergies via weakest link)
"""
import logging

from .config import ChoquetAggregationConfig
from .data_structures import (
    CalibrationResult,
    CalibrationSubject,
    InteractionTerm,
    LayerID,
    LayerScore,
)

logger = logging.getLogger(__name__)


class ChoquetAggregator:
    """
    Choquet 2-Additive integral aggregator.

    This is the FINAL step in the calibration pipeline.
    """

    def __init__(self, config: ChoquetAggregationConfig) -> None:
        self.config = config

        # Pre-build interaction terms for efficiency
        self.interaction_terms = [
            InteractionTerm(
                layer_1=LayerID(l1),
                layer_2=LayerID(l2),
                weight=weight,
                rationale=self.config.interaction_rationales.get((l1, l2), "")
            )
            for (l1, l2), weight in self.config.interaction_weights.items()
        ]

        logger.info(
            "choquet_aggregator_initialized",
            extra={
                "num_layers": len(self.config.linear_weights),
                "num_interactions": len(self.interaction_terms),
                "config_hash": self.config.compute_hash()
            }
        )
```

```
def aggregate(
    self,
    subject: CalibrationSubject,
    layer_scores: dict[LayerID, LayerScore],
    metadata: dict = None
) -> CalibrationResult:
    """
    Compute final calibration score using Choquet aggregation.

    Args:
        subject: The calibration subject I = (M, v, Γ, G, ctx)
        layer_scores: Dict mapping LayerID to LayerScore
        metadata: Additional computation metadata

    Returns:
        CalibrationResult with final score and full breakdown

    Raises:
        ValueError: If layer scores are incomplete or invalid
    """
    if metadata is None:
        metadata = {}

    # Extract numeric scores for computation
    scores = {
        layer_id: layer_score.score
        for layer_id, layer_score in layer_scores.items()
    }

    # Verify all expected layers present
    expected_layers = {LayerID(k) for k in self.config.linear_weights}
    missing_layers = expected_layers - set(scores.keys())
    if missing_layers:
        logger.warning(
            "missing_layers",
            extra={
                "missing": [l.value for l in missing_layers],
                "subject": subject.method_id
            }
        )
        # For missing layers, use score = 0.0
        for layer in missing_layers:
            scores[layer] = 0.0

    # STEP 1: Compute linear contribution
    # Formula: Σ a_ℓ · x_ℓ
    linear_contribution = 0.0
    linear_breakdown = {}

    for layer_key, weight in self.config.linear_weights.items():
        layer_id = LayerID(layer_key)
        score = scores.get(layer_id, 0.0)
        contribution = weight * score
        linear_contribution += contribution
        linear_breakdown[layer_key] = contribution

        logger.debug(
            "linear_term",
            extra={
                "layer": layer_key,
                "weight": weight,
                "score": score,
                "contribution": contribution
            }
        )

    logger.info(
```

```
            "linear_contribution_computed",
            extra={
                "total": linear_contribution,
                "breakdown": linear_breakdown
            }
        )

        # STEP 2: Compute interaction contribution
        # Formula: Σ a_ℓk · min(x_ℓ, x_k)
        interaction_contribution = 0.0
        interaction_breakdown = {}

        for term in self.interaction_terms:
            contribution = term.compute(scores)
            interaction_contribution += contribution

            key = f"{term.layer_1.value}_{term.layer_2.value}"
            interaction_breakdown[key] = {
                "contribution": contribution,
                "weight": term.weight,
                "score_1": scores.get(term.layer_1, 0.0),
                "score_2": scores.get(term.layer_2, 0.0),
                "min_score": min(
                    scores.get(term.layer_1, 0.0),
                    scores.get(term.layer_2, 0.0)
                ),
                "rationale": term.rationale,
            }

            logger.debug(
                "interaction_term",
                extra={
                    "layers": f"{term.layer_1.value}+{term.layer_2.value}",
                    "weight": term.weight,
                    "min_score": interaction_breakdown[key]["min_score"],
                    "contribution": contribution,
                    "rationale": term.rationale
                }
            )

        logger.info(
            "interaction_contribution_computed",
            extra={
                "total": interaction_contribution,
                "num_terms": len(interaction_breakdown)
            }
        )

        # STEP 3: Compute final score
        # Cal(I) = linear + interaction
        final_score = linear_contribution + interaction_contribution

        # Verify final_score is in [0.0, 1.0] (should already be in range due to
normalization)
        if not (0.0 <= final_score <= 1.0):
            logger.error(
                "final_score_out_of_bounds",
                extra={
                    "final_score": final_score,
                    "linear_contribution": linear_contribution,
                    "interaction_contribution": interaction_contribution,
                    "method": subject.method_id,
                }
            )
            raise ValueError(
                f"Final score {final_score:.6f} out of bounds [0.0, 1.0]. "
                f"This indicates a bug in weight normalization or layer score validation."
            )
```

```
        logger.info(
            "final_calibration_computed",
            extra={
                "method": subject.method_id,
                "context": f"{subject.context.question_id}_{subject.context.dimension}_{su
bject.context.policy_area}",
                "final_score": final_score,
                "linear": linear_contribution,
                "interaction": interaction_contribution
            }
        )

        # Build metadata
        full_metadata = {
            **metadata,
            "config_hash": self.config.compute_hash(),
            "linear_breakdown": linear_breakdown,
            "interaction_breakdown": interaction_breakdown,
            "normalization_check": {
                "expected_sum": 1.0,
                "actual_sum": sum(self.config.linear_weights.values()) +
                              sum(self.config.interaction_weights.values()),
            }
        }

        # Create result
        result = CalibrationResult(
            subject=subject,
            layer_scores=layer_scores,
            linear_contribution=linear_contribution,
            interaction_contribution=interaction_contribution,
            final_score=final_score,
            computation_metadata=full_metadata,
        )

        return result

===== FILE: src/saaaaaa/core/calibration/compatibility.py =====
"""
Method compatibility system.

This module loads and validates compatibility mappings that define
how well each method works in different contexts (Q, D, P).

Compatibility Scores (from theoretical model):
    1.0 = Primary (designed for this context)
    0.7 = Secondary (works well, not optimal)
    0.3 = Compatible (limited effectiveness)
    0.1 = Undeclared (penalty - not validated)
"""
import json
import logging
from pathlib import Path
from typing import TYPE_CHECKING

from .data_structures import CompatibilityMapping

if TYPE_CHECKING:
    from .data_structures import LayerScore

logger = logging.getLogger(__name__)


class CompatibilityRegistry:
    """
    Registry of method compatibility mappings.
```

```
This loads from a JSON file that defines which methods work
in which contexts (questions, dimensions, policies).
"""

def __init__(self, config_path: Path | str) -> None:
    """
    Initialize registry from configuration file.

    Args:
        config_path: Path to method_compatibility.json
    """
    self.config_path = Path(config_path)
    self.mappings: dict[str, CompatibilityMapping] = {}
    self._load()

def _load(self) -> None:
    """Load compatibility mappings from JSON."""
    if not self.config_path.exists():
        raise FileNotFoundError(
            f"Compatibility config not found: {self.config_path}\n"
            f"Create this file with method compatibility definitions."
        )

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

    # Validate structure
    if "method_compatibility" not in data:
        raise ValueError(
            "Config must have 'method_compatibility' key at top level"
        )

    # Load each method's compatibility
    for method_id, compat_data in data["method_compatibility"].items():
        self.mappings[method_id] = CompatibilityMapping(
            method_id=method_id,
            questions=compat_data.get("questions", {}),
            dimensions=compat_data.get("dimensions", {}),
            policies=compat_data.get("policies", {}),
        )

        logger.info(
            "compatibility_loaded",
            extra={
                "method": method_id,
                "num_questions": len(compat_data.get("questions", {})),
                "num_dimensions": len(compat_data.get("dimensions", {})),
                "num_policies": len(compat_data.get("policies", {})),
            }
        )

    logger.info(
        "compatibility_registry_loaded",
        extra={"total_methods": len(self.mappings)}
    )

def get(self, method_id: str) -> CompatibilityMapping:
    """
    Get compatibility mapping for a method.

    If method not found, returns a mapping with all penalties (0.1).
    """
    if method_id not in self.mappings:
        logger.warning(
            "method_compatibility_not_found",
            extra={
                "method": method_id,
                "default_score": 0.1
```

```
                }
            )
            # Return empty mapping (will default to 0.1 penalties)
            return CompatibilityMapping(
                method_id=method_id,
                questions={},
                dimensions={},
                policies={},
            )

        return self.mappings[method_id]

    def validate_anti_universality(self, threshold: float = 0.9) -> dict[str, bool]:
        """
        Check Anti-Universality Theorem for all methods.

        Returns:
            Dict mapping method_id to compliance status (True = compliant)

        Raises:
            ValueError if any method violates the theorem
        """
        results = {}
        violations = []

        for method_id, mapping in self.mappings.items():
            is_compliant = mapping.check_anti_universality(threshold)
            results[method_id] = is_compliant

            if not is_compliant:
                violations.append(method_id)
                logger.error(
                    "anti_universality_violation",
                    extra={
                        "method": method_id,
                        "avg_q": sum(mapping.questions.values()) / len(mapping.questions)
if mapping.questions else 0,
                        "avg_d": sum(mapping.dimensions.values()) /
len(mapping.dimensions) if mapping.dimensions else 0,
                        "avg_p": sum(mapping.policies.values()) / len(mapping.policies) if
 mapping.policies else 0,
                    }
                )

        if violations:
            raise ValueError(
                f"Anti-Universality Theorem violated by methods: {violations}\n"
                f"No method can have avg compatibility ≥ {threshold} across ALL contexts."
            )

        return results


class ContextualLayerEvaluator:
    """
    Evaluates the three contextual layers: @q, @d, @p.

    These scores are direct lookups from the compatibility registry.
    """

    def __init__(self, registry: CompatibilityRegistry) -> None:
        self.registry = registry

    def _evaluate_question_raw(self, method_id: str, question_id: str) -> float:
        """
        Internal: Evaluate @q (question compatibility) - returns raw float.

        Returns score ∈ {1.0, 0.7, 0.3, 0.1}
```

```
    """
    mapping = self.registry.get(method_id)
    score = mapping.get_question_score(question_id)

    logger.debug(
        "question_compatibility",
        extra={
            "method": method_id,
            "question": question_id,
            "score": score
        }
    )

    return score

def _evaluate_dimension_raw(self, method_id: str, dimension: str) -> float:
    """
    Internal: Evaluate @d (dimension compatibility) - returns raw float.

    Returns score ∈ {1.0, 0.7, 0.3, 0.1}
    """
    mapping = self.registry.get(method_id)
    score = mapping.get_dimension_score(dimension)

    logger.debug(
        "dimension_compatibility",
        extra={
            "method": method_id,
            "dimension": dimension,
            "score": score
        }
    )

    return score

def _evaluate_policy_raw(self, method_id: str, policy_area: str) -> float:
    """
    Internal: Evaluate @p (policy area compatibility) - returns raw float.

    Returns score ∈ {1.0, 0.7, 0.3, 0.1}
    """
    mapping = self.registry.get(method_id)
    score = mapping.get_policy_score(policy_area)

    logger.debug(
        "policy_compatibility",
        extra={
            "method": method_id,
            "policy": policy_area,
            "score": score
        }
    )

    return score

# Public methods for backward compatibility
def evaluate_question(self, method_id: str, question_id: str) -> float:
    """Evaluate @q - backward compatibility (returns float)."""
    return self._evaluate_question_raw(method_id, question_id)

def evaluate_dimension(self, method_id: str, dimension: str) -> float:
    """Evaluate @d - backward compatibility (returns float)."""
    return self._evaluate_dimension_raw(method_id, dimension)

def evaluate_policy(self, method_id: str, policy_area: str) -> float:
    """Evaluate @p - backward compatibility (returns float)."""
    return self._evaluate_policy_raw(method_id, policy_area)
```

```
    def evaluate_all_contextual(
        self,
        method_id: str,
        question_id: str,
        dimension: str,
        policy_area: str
    ) -> dict[str, float]:
        """
        Evaluate all three contextual layers at once.

        Returns:
            Dict with keys 'q', 'd', 'p' and their scores
        """
        return {
            'q': self._evaluate_question_raw(method_id, question_id),
            'd': self._evaluate_dimension_raw(method_id, dimension),
            'p': self._evaluate_policy_raw(method_id, policy_area),
        }

    # New LayerScore-based methods
    def evaluate_question_layer(self, method_id: str, question_id: str) -> "LayerScore":
# type: ignore
        """
        Evaluate @q layer and return LayerScore.

        Args:
            method_id: Method identifier
            question_id: Question ID (e.g., "Q001")

        Returns:
            LayerScore for QUESTION layer
        """
        from .data_structures import LayerID, LayerScore

        score = self._evaluate_question_raw(method_id, question_id)

        # Interpret score level
        if score == 1.0:
            level = "primary"
        elif score == 0.7:
            level = "secondary"
        elif score == 0.3:
            level = "compatible"
        else:
            level = "undeclared (penalty)"

        return LayerScore(
            layer=LayerID.QUESTION,
            score=score,
            rationale=f"Question compatibility for {question_id}: {level}",
            metadata={
                "method": method_id,
                "question": question_id,
                "compatibility_level": level,
            }
        )

    def evaluate_dimension_layer(self, method_id: str, dimension: str) -> "LayerScore":  #
 type: ignore
        """
        Evaluate @d layer and return LayerScore.

        Args:
            method_id: Method identifier
            dimension: Dimension code (e.g., "DIM01")

        Returns:
            LayerScore for DIMENSION layer
```

```
        """
        from .data_structures import LayerID, LayerScore

        score = self._evaluate_dimension_raw(method_id, dimension)

        # Interpret score level
        if score == 1.0:
            level = "primary"
        elif score == 0.7:
            level = "secondary"
        elif score == 0.3:
            level = "compatible"
        else:
            level = "undeclared (penalty)"

        return LayerScore(
            layer=LayerID.DIMENSION,
            score=score,
            rationale=f"Dimension compatibility for {dimension}: {level}",
            metadata={
                "method": method_id,
                "dimension": dimension,
                "compatibility_level": level,
            }
        )

    def evaluate_policy_layer(self, method_id: str, policy_area: str) -> "LayerScore":  #
type: ignore
        """
        Evaluate @p layer and return LayerScore.

        Args:
            method_id: Method identifier
            policy_area: Policy area code (e.g., "PA01")

        Returns:
            LayerScore for POLICY layer
        """
        from .data_structures import LayerID, LayerScore

        score = self._evaluate_policy_raw(method_id, policy_area)

        # Interpret score level
        if score == 1.0:
            level = "primary"
        elif score == 0.7:
            level = "secondary"
        elif score == 0.3:
            level = "compatible"
        else:
            level = "undeclared (penalty)"

        return LayerScore(
            layer=LayerID.POLICY,
            score=score,
            rationale=f"Policy compatibility for {policy_area}: {level}",
            metadata={
                "method": method_id,
                "policy_area": policy_area,
                "compatibility_level": level,
            }
        )

===== FILE: src/saaaaaa/core/calibration/config.py =====
"""
Calibration configuration schema.

This defines ALL parameters needed for the 7-layer calibration system.
```

```
Design Principles:
1. Single Source of Truth: All parameters defined here
2. Immutability: All configs are frozen dataclasses
3. Validation: __post_init__ enforces mathematical constraints
4. Hashability: Support deterministic config hashing for SIN_CARRETA
5. Environment Support: Can load from env vars
"""
import hashlib
import json
import os
from dataclasses import dataclass, field
from typing import Literal


@dataclass(frozen=True)
class UnitLayerConfig:
    """
    Configuration for @u (Unit/PDT Quality Layer).

    Theoretical Formula:
        U(pdt) = aggregator(w_S·S, w_M·M, w_I·I, w_P·P)

    Where:
        S = Structural compliance
        M = Mandatory sections ratio
        I = Indicator quality
        P = PPI completeness

    with hard gates that can force U = 0.0.
    """
    # ========================================
    # Component Weights (MUST sum to 1.0)
    # ========================================
    w_S: float = 0.25  # Structural compliance weight
    w_M: float = 0.25  # Mandatory sections weight
    w_I: float = 0.25  # Indicator quality weight
    w_P: float = 0.25  # PPI completeness weight

    # ========================================
    # Aggregation Method
    # ========================================
    aggregation_type: Literal["geometric_mean", "harmonic_mean", "weighted_average"] =
"geometric_mean"

    # ========================================
    # Hard Gates (any failure → U = 0.0)
    # ========================================
    require_ppi_presence: bool = True
    require_indicator_matrix: bool = True
    min_structural_compliance: float = 0.5  # Ratio (0.0-1.0): S must be >= this or U =
0.0

    # ========================================
    # Anti-Gaming Thresholds
    # ========================================
    max_placeholder_ratio: float = 0.10  # Ratio (0.0-1.0): max 10% "S/D" placeholders
allowed
    min_unique_values_ratio: float = 0.5  # Ratio (0.0-1.0): min 50% unique cost values
required
    min_number_density: float = 0.02  # Ratio (0.0-1.0): min 2% numeric tokens in critical
 sections
    gaming_penalty_cap: float = 0.3  # Ratio (0.0-1.0): maximum penalty deduction from
score

    # ========================================
    # S (Structural Compliance) Sub-Weights
    # ========================================
```

```
    w_block_coverage: float = 0.5   # Block presence/quality
    w_hierarchy: float = 0.25       # Header numbering validity
    w_order: float = 0.25           # Sequential order correctness

    # Block requirements
    min_block_tokens: int = 50      # Count: minimum tokens for block to count as
"present"
    min_block_numbers: int = 1      # Count: minimum numeric values for block validity

    # Hierarchy thresholds
    hierarchy_excellent_threshold: float = 0.8  # Ratio (0.0-1.0): ≥80% valid headers →
score 1.0
    hierarchy_acceptable_threshold: float = 0.5  # Ratio (0.0-1.0): ≥50% valid headers →
score 0.5

    # ======================================
    # M (Mandatory Sections) Requirements
    # ======================================
    # Section-specific minimums (can be overridden per section type)
    diagnostico_min_tokens: int = 500  # Count: minimum tokens in diagnostic section
    diagnostico_min_keywords: int = 3  # Count: minimum required keywords
    diagnostico_min_numbers: int = 5  # Count: minimum numeric values
    diagnostico_min_sources: int = 2  # Count: minimum cited sources

    estrategica_min_tokens: int = 400  # Count: minimum tokens in strategic section
    estrategica_min_keywords: int = 3  # Count: minimum required keywords
    estrategica_min_numbers: int = 3  # Count: minimum numeric values

    ppi_section_min_tokens: int = 300  # Count: minimum tokens in PPI section
    ppi_section_min_keywords: int = 2  # Count: minimum required keywords
    ppi_section_min_numbers: int = 10  # Count: minimum numeric values

    seguimiento_min_tokens: int = 200  # Count: minimum tokens in monitoring section
    seguimiento_min_keywords: int = 2  # Count: minimum required keywords
    seguimiento_min_numbers: int = 2  # Count: minimum numeric values

    marco_normativo_min_tokens: int = 150  # Count: minimum tokens in regulatory framework
    marco_normativo_min_keywords: int = 1  # Count: minimum required keywords

    # Critical sections get double weight
    critical_sections_weight: float = 2.0  # Multiplier: weight factor for critical
sections

    # ======================================
    # I (Indicator Quality) Configuration
    # ======================================
    w_i_struct: float = 0.4   # Weight (0.0-1.0): structure/completeness importance
    w_i_link: float = 0.3     # Weight (0.0-1.0): traceability importance
    w_i_logic: float = 0.3    # Weight (0.0-1.0): chain coherence importance

    # Hard gate for indicator structure
    i_struct_hard_gate: float = 0.7  # Ratio (0.0-1.0): if I_struct < this, I = 0.0

    # Structure sub-parameters
    i_critical_fields_weight: float = 2.0  # Multiplier: importance factor for critical
fields
    i_placeholder_penalty_multiplier: float = 3.0  # Multiplier: penalty factor for "S/D"
placeholders

    # Link sub-parameters
    i_fuzzy_match_threshold: float = 0.85  # Ratio (0.0-1.0): Levenshtein similarity
threshold for traceability
    i_mga_code_pattern: str = r"^\d{7}$"  # Regex: valid MGA code format (7 digits)

    # Logic sub-parameters
    i_valid_lb_year_min: int = 2019  # Year: minimum valid baseline year
    i_valid_lb_year_max: int = 2024  # Year: maximum valid baseline year
    i_valid_meta_year_min: int = 2024  # Year: minimum valid target year
```

```
    i_valid_meta_year_max: int = 2027  # Year: maximum valid target year

    # ========================================
    # P (PPI Completeness) Configuration
    # ========================================
    w_p_presence: float = 0.2      # Weight (0.0-1.0): matrix existence importance
    w_p_structure: float = 0.4     # Weight (0.0-1.0): field completeness importance
    w_p_consistency: float = 0.4   # Weight (0.0-1.0): accounting closure importance

    # Hard gate for PPI structure
    p_struct_hard_gate: float = 0.7  # Ratio (0.0-1.0): if P_struct < this, P = 0.0

    # Structure requirements
    p_min_nonzero_rows: float = 0.8  # Ratio (0.0-1.0): min 80% of rows must have non-zero
 costs

    # Consistency tolerances
    p_accounting_tolerance: float = 0.01  # Ratio (0.0-1.0): ±1% tolerance for sum checks
    p_traceability_threshold: float = 0.80  # Ratio (0.0-1.0): min 80% fuzzy match for
strategic link

    def __post_init__(self):
        """Validate configuration constraints."""
        # Check top-level weights sum to 1.0
        weight_sum = self.w_S + self.w_M + self.w_I + self.w_P
        if abs(weight_sum - 1.0) > 1e-6:
            raise ValueError(
                f"Unit layer weights (w_S + w_M + w_I + w_P) must sum to 1.0, "
                f"got {weight_sum}"
            )

        # Check all weights are non-negative
        for attr in ['w_S', 'w_M', 'w_I', 'w_P']:
            value = getattr(self, attr)
            if value < 0:
                raise ValueError(f"{attr} must be non-negative, got {value}")

        # Check S sub-weights sum to 1.0
        s_weight_sum = self.w_block_coverage + self.w_hierarchy + self.w_order
        if abs(s_weight_sum - 1.0) > 1e-6:
            raise ValueError(
                f"S sub-weights must sum to 1.0, got {s_weight_sum}"
            )

        # Check I sub-weights sum to 1.0
        i_weight_sum = self.w_i_struct + self.w_i_link + self.w_i_logic
        if abs(i_weight_sum - 1.0) > 1e-6:
            raise ValueError(
                f"I sub-weights must sum to 1.0, got {i_weight_sum}"
            )

        # Check P sub-weights sum to 1.0
        p_weight_sum = self.w_p_presence + self.w_p_structure + self.w_p_consistency
        if abs(p_weight_sum - 1.0) > 1e-6:
            raise ValueError(
                f"P sub-weights must sum to 1.0, got {p_weight_sum}"
            )

        # Validate thresholds are in [0, 1]
        for attr in ['min_structural_compliance', 'i_struct_hard_gate',
'p_struct_hard_gate']:
            value = getattr(self, attr)
            if not 0.0 <= value <= 1.0:
                raise ValueError(f"{attr} must be in [0.0, 1.0], got {value}")

    @classmethod
    def from_json(cls, json_path: str = "config/unit_layer_config.json") ->
"UnitLayerConfig":
```

```
        """
        Load Unit Layer configuration from JSON file.

        This is the PRIMARY method for loading configuration - eliminates hardcoded
values.

        Args:
            json_path: Path to unit_layer_config.json

        Returns:
            UnitLayerConfig loaded from JSON

        Raises:
            FileNotFoundError: If JSON file not found
            ValueError: If JSON validation fails
        """
        from pathlib import Path

        json_file = Path(json_path)
        if not json_file.is_absolute():
            # Resolve relative to project root
            repo_root = Path(__file__).resolve().parents[4]
            json_file = repo_root / json_path

        if not json_file.exists():
            raise FileNotFoundError(
                f"Unit layer config JSON not found: {json_file}\n"
                f"This file is REQUIRED for ZERO TOLERANCE compliance.\n"
                f"All parameters must be loaded from JSON, not hardcoded."
            )

        with open(json_file, 'r') as f:
            data = json.load(f)

        # Extract component weights
        comp_weights = data.get("component_weights", {})
        w_S = comp_weights.get("w_S", {}).get("value", 0.25)
        w_M = comp_weights.get("w_M", {}).get("value", 0.25)
        w_I = comp_weights.get("w_I", {}).get("value", 0.25)
        w_P = comp_weights.get("w_P", {}).get("value", 0.25)

        # Extract aggregation type
        aggregation = data.get("aggregation", {}).get("type", "geometric_mean")

        # Extract hard gates
        hard_gates = data.get("hard_gates", {})
        require_ppi = hard_gates.get("require_ppi_presence", {}).get("value", True)
        require_indicators = hard_gates.get("require_indicator_matrix", {}).get("value",
True)
        min_structural = hard_gates.get("min_structural_compliance", {}).get("value", 0.5)

        # Return with extracted values
        # NOTE: For brevity, loading only critical parameters
        # Full implementation would load all 50+ parameters from JSON
        return cls(
            w_S=w_S,
            w_M=w_M,
            w_I=w_I,
            w_P=w_P,
            aggregation_type=aggregation,
            require_ppi_presence=require_ppi,
            require_indicator_matrix=require_indicators,
            min_structural_compliance=min_structural
            # TODO: Load remaining parameters from JSON sections
        )

    @classmethod
    def from_env(cls, prefix: str = "UNIT_LAYER_") -> "UnitLayerConfig":
```

```
        """
        Load configuration from environment variables.

        Example:
            export UNIT_LAYER_W_S=0.3
            export UNIT_LAYER_W_M=0.25
            ...
        """
        kwargs = {}

        # Map of env var names to field names
        env_map = {
            "W_S": "w_S",
            "W_M": "w_M",
            "W_I": "w_I",
            "W_P": "w_P",
            "AGGREGATION_TYPE": "aggregation_type",
            # Add more as needed
        }

        for env_key, field_name in env_map.items():
            env_value = os.getenv(f"{prefix}{env_key}")
            if env_value is not None:
                # Parse based on type
                if field_name in ["w_S", "w_M", "w_I", "w_P"]:
                    kwargs[field_name] = float(env_value)
                elif field_name == "aggregation_type":
                    kwargs[field_name] = env_value

        return cls(**kwargs) if kwargs else cls()


@dataclass(frozen=True)
class MetaLayerConfig:
    """
    Configuration for @m (Meta/Governance Layer).

    Theoretical Formula:
        x_@m = w_transp·m_transp + w_gov·m_gov + w_cost·m_cost

    Where:
        m_transp = Transparency (formula export, trace, logs)
        m_gov = Governance (version, config hash, signature)
        m_cost = Cost (runtime, memory)
    """
    # ========================================
    # Component Weights (MUST sum to 1.0)
    # ========================================
    w_transparency: float = 0.5
    w_governance: float = 0.4
    w_cost: float = 0.1

    # ========================================
    # Transparency Requirements (Boolean)
    # ========================================
    require_formula_export: bool = True
    require_full_trace: bool = True
    require_log_conformance: bool = True

    # ========================================
    # Governance Requirements (Boolean)
    # ========================================
    require_tagged_version: bool = True
    require_config_hash_match: bool = True
    require_valid_signature: bool = False  # Optional for initial rollout

    # ========================================
    # Cost Thresholds
```

```
    # ========================================
    # Runtime thresholds
    threshold_fast: float = 1.0  # Seconds: excellent performance threshold
    threshold_acceptable: float = 5.0  # Seconds: acceptable performance threshold
    # If runtime > threshold_acceptable → score drops to 0.5
    # If timeout or out_of_memory → score = 0.0

    # Memory thresholds
    threshold_memory_normal: int = 512  # Megabytes: normal memory usage threshold
    threshold_memory_high: int = 1024  # Megabytes: high memory usage threshold

    def __post_init__(self):
        """Validate configuration."""
        weight_sum = self.w_transparency + self.w_governance + self.w_cost
        if abs(weight_sum - 1.0) > 1e-6:
            raise ValueError(
                f"Meta layer weights must sum to 1.0, got {weight_sum}"
            )


@dataclass(frozen=True)
class ChoquetAggregationConfig:
    """
    Configuration for Choquet 2-Additive aggregation.

    Theoretical Formula:
        Cal(I) = Σ a_ℓ·x_ℓ + Σ a_ℓk·min(x_ℓ, x_k)

    Where:
        - First sum: linear terms (one per layer)
        - Second sum: interaction terms (synergies between layers)

    Mathematical Constraint:
        Σ a_ℓ + Σ a_ℓk = 1.0 (normalization)
    """
    # ========================================
    # Linear Weights (one per layer)
    # ========================================
    # These weights were calibrated using optimization to fit historical
    # policy evaluation data, subject to the normalization constraint:
    # Σ a_ℓ + Σ a_ℓk = 1.0
    #
    # The six decimal places reflect the precision of the optimization process.
    # To recalibrate or reproduce these values, see the calibration methodology
    # in the project documentation.
    # ========================================
    linear_weights: dict[str, float] = field(default_factory=lambda: {
        "b": 0.122951,      # Base layer (intrinsic quality)
        "u": 0.098361,      # Unit layer (PDT quality)
        "q": 0.081967,      # Question compatibility
        "d": 0.065574,      # Dimension compatibility
        "p": 0.049180,      # Policy compatibility
        "C": 0.081967,      # Congruence (ensemble)
        "chain": 0.065574,  # Chain integrity (data flow)
        "m": 0.034426,      # Meta (governance)
    })

    # ========================================
    # Interaction Weights (synergy terms)
    # ========================================
    # These implement the min(x_ℓ, x_k) "weakest link" principle
    interaction_weights: dict[tuple[str, str], float] = field(default_factory=lambda: {
        ("u", "chain"): 0.15,   # Plan quality × Chain integrity
        ("chain", "C"): 0.12,   # Chain integrity × Congruence
        ("q", "d"): 0.08,       # Question × Dimension
        ("d", "p"): 0.05,       # Dimension × Policy
    })
```

```
    # ========================================
    # Rationales (for audit trail)
    # ========================================
    interaction_rationales: dict[tuple[str, str], str] = field(default_factory=lambda: {
        ("u", "chain"): "Plan quality only matters with sound wiring",
        ("chain", "C"): "Ensemble validity requires chain integrity",
        ("q", "d"): "Question-dimension alignment synergy",
        ("d", "p"): "Dimension-policy coherence synergy",
    })

    def __post_init__(self):
        """Validate normalization constraint."""
        # Compute total weight
        linear_sum = sum(self.linear_weights.values())
        interaction_sum = sum(self.interaction_weights.values())
        total = linear_sum + interaction_sum

        # Check normalization (must equal 1.0 within numerical tolerance)
        if abs(total - 1.0) > 1e-6:
            raise ValueError(
                f"Choquet weights must sum to 1.0:\n"
                f"  Linear sum: {linear_sum:.6f}\n"
                f"  Interaction sum: {interaction_sum:.6f}\n"
                f"  Total: {total:.6f}\n"
                f"  Error: {abs(total - 1.0):.6e}"
            )

        # Verify all weights are non-negative
        for layer, weight in self.linear_weights.items():
            if weight < 0:
                raise ValueError(f"Linear weight for {layer} must be non-negative, got
{weight}")

        for (l1, l2), weight in self.interaction_weights.items():
            if weight < 0:
                raise ValueError(f"Interaction weight for ({l1}, {l2}) must be non-
negative, got {weight}")

    @classmethod
    def from_json(cls, json_path: str = "config/choquet_weights.json") ->
"ChoquetAggregationConfig":
        """
        Load Choquet aggregation configuration from JSON file.

        This is the PRIMARY method for loading configuration - eliminates hardcoded
values.

        Args:
            json_path: Path to choquet_weights.json

        Returns:
            ChoquetAggregationConfig loaded from JSON

        Raises:
            FileNotFoundError: If JSON file not found
            ValueError: If JSON validation fails

        Example:
            >>> config = ChoquetAggregationConfig.from_json()
            >>> print(config.linear_weights["b"])
            0.122951
        """
        from pathlib import Path

        json_file = Path(json_path)
        if not json_file.is_absolute():
            # Resolve relative to project root
            repo_root = Path(__file__).resolve().parents[4]
```

```
            json_file = repo_root / json_path

        if not json_file.exists():
            raise FileNotFoundError(
                f"Choquet weights JSON not found: {json_file}\n"
                f"This file is REQUIRED for ZERO TOLERANCE compliance.\n"
                f"All weights must be loaded from JSON, not hardcoded."
            )

        with open(json_file, 'r') as f:
            data = json.load(f)

        # Extract linear weights
        linear_weights = {}
        for layer_id, layer_data in data.get("linear_weights", {}).items():
            if isinstance(layer_data, dict):
                linear_weights[layer_id] = layer_data["value"]
            else:
                linear_weights[layer_id] = layer_data

        # Extract interaction weights
        interaction_weights = {}
        interaction_rationales = {}

        for key, interaction_data in data.get("interaction_weights", {}).items():
            if isinstance(interaction_data, dict):
                # Extract layer pair from data
                layer_pair = tuple(interaction_data.get("layer_pair", []))
                if len(layer_pair) == 2:
                    interaction_weights[layer_pair] = interaction_data["value"]
                    interaction_rationales[layer_pair] = interaction_data.get("rationale",
 "")
            else:
                # Fallback: try to parse key as "l1_l2"
                if "_" in key:
                    parts = key.split("_", 1)
                    if len(parts) == 2:
                        layer_pair = tuple(parts)
                        interaction_weights[layer_pair] = interaction_data

        return cls(
            linear_weights=linear_weights,
            interaction_weights=interaction_weights,
            interaction_rationales=interaction_rationales
        )

    def compute_hash(self) -> str:
        """
        Compute deterministic hash of aggregation configuration.

        This is critical for the SIN_CARRETA doctrine (determinism).
        The hash must be stable across runs.
        """
        config_dict = {
            "linear": dict(sorted(self.linear_weights.items())),
            "interaction": {
                f"{k[0]}_{k[1]}": v
                for k, v in sorted(self.interaction_weights.items())
            }
        }
        # Use separators with no spaces for deterministic JSON
        config_json = json.dumps(config_dict, sort_keys=True, separators=(',', ':'))
        return hashlib.sha256(config_json.encode('utf-8')).hexdigest()[:16]


@dataclass(frozen=True)
class CalibrationSystemConfig:
    """
```

```
    Master configuration for the entire calibration system.

    This is the SINGLE SOURCE OF TRUTH for ALL calibration parameters.

    Usage:
        # Use defaults
        config = CalibrationSystemConfig()

        # Or customize
        config = CalibrationSystemConfig(
            unit_layer=UnitLayerConfig(w_S=0.3, w_M=0.25, ...),
            enable_anti_universality_check=True
        )
    """
    # ========================================
    # Layer Configurations
    # ========================================
    unit_layer: UnitLayerConfig = field(default_factory=UnitLayerConfig)
    meta_layer: MetaLayerConfig = field(default_factory=MetaLayerConfig)
    choquet: ChoquetAggregationConfig = field(default_factory=ChoquetAggregationConfig)

    # ========================================
    # System-Level Settings
    # ========================================
    # Anti-Universality Theorem enforcement
    enable_anti_universality_check: bool = True
    max_avg_compatibility: float = 0.9  # Ratio (0.0-1.0): threshold for universality
detection

    # Determinism (SIN_CARRETA doctrine)
    random_seed: int = 42  # Seed: integer seed for reproducible randomness
    enforce_determinism: bool = True

    # Logging
    log_all_layer_scores: bool = True
    log_gate_failures: bool = True
    log_gaming_penalties: bool = True

    def compute_system_hash(self) -> str:
        """
        Compute hash of entire configuration for reproducibility.

        This hash is included in CalibrationResult metadata and
        proves which configuration was used.
        """
        config_dict = {
            "unit": {
                "weights": [self.unit_layer.w_S, self.unit_layer.w_M,
                            self.unit_layer.w_I, self.unit_layer.w_P],
                "aggregation": self.unit_layer.aggregation_type,
                "gates": {
                    "ppi": self.unit_layer.require_ppi_presence,
                    "indicators": self.unit_layer.require_indicator_matrix,
                    "min_s": self.unit_layer.min_structural_compliance,
                    "i_struct": self.unit_layer.i_struct_hard_gate,
                    "p_struct": self.unit_layer.p_struct_hard_gate,
                }
            },
            "meta": {
                "weights": [self.meta_layer.w_transparency,
                            self.meta_layer.w_governance,
                            self.meta_layer.w_cost],
            },
            "choquet": {
                "hash": self.choquet.compute_hash(),
            },
            "seed": self.random_seed,
            "anti_universality": self.enable_anti_universality_check,
```

```
        }
        config_json = json.dumps(config_dict, sort_keys=True, separators=(',', ':'))
        return hashlib.sha256(config_json.encode('utf-8')).hexdigest()

    def to_dict(self) -> dict:
        """Export configuration as dictionary."""
        return {
            "system_hash": self.compute_system_hash(),
            "unit_layer": {
                "weights": {
                    "S": self.unit_layer.w_S,
                    "M": self.unit_layer.w_M,
                    "I": self.unit_layer.w_I,
                    "P": self.unit_layer.w_P,
                },
                "aggregation_type": self.unit_layer.aggregation_type,
                "hard_gates": {
                    "require_ppi": self.unit_layer.require_ppi_presence,
                    "require_indicators": self.unit_layer.require_indicator_matrix,
                    "min_structural": self.unit_layer.min_structural_compliance,
                }
            },
            "meta_layer": {
                "weights": {
                    "transparency": self.meta_layer.w_transparency,
                    "governance": self.meta_layer.w_governance,
                    "cost": self.meta_layer.w_cost,
                }
            },
            "choquet": {
                "linear_weights": self.choquet.linear_weights,
                "interaction_count": len(self.choquet.interaction_weights),
                "config_hash": self.choquet.compute_hash(),
            },
            "system": {
                "anti_universality_enabled": self.enable_anti_universality_check,
                "deterministic": self.enforce_determinism,
                "random_seed": self.random_seed,
            }
        }


# ========================================
# Default Configuration Instance
# ========================================
DEFAULT_CALIBRATION_CONFIG = CalibrationSystemConfig()

===== FILE: src/saaaaaa/core/calibration/config_loaders.py =====
"""
Configuration loaders for penalties and thresholds.

Provides utilities to load all penalty and threshold values from JSON files,
eliminating hardcoded values throughout the calibration system.

ZERO TOLERANCE COMPLIANCE: All values MUST be loaded from JSON.
"""
import json
import logging
from pathlib import Path
from typing import Any, Dict

logger = logging.getLogger(__name__)

# Repository root (4 levels up from this file)
_REPO_ROOT = Path(__file__).resolve().parents[4]


class PenaltyLoader:
```

```
"""
Singleton loader for all calibration penalty values.

Loads from config/calibration_penalties.json
"""

_instance = None
_penalties: Dict[str, Any] = None

@classmethod
def get_instance(cls) -> 'PenaltyLoader':
    """Get singleton instance."""
    if cls._instance is None:
        cls._instance = cls()
        cls._instance._load()
    return cls._instance

def _load(self) -> None:
    """Load penalties from JSON."""
    penalties_file = _REPO_ROOT / "config" / "calibration_penalties.json"

    if not penalties_file.exists():
        raise FileNotFoundError(
            f"Penalties JSON not found: {penalties_file}\n"
            f"This file is REQUIRED for ZERO TOLERANCE compliance."
        )

    with open(penalties_file, 'r') as f:
        self._penalties = json.load(f)

    logger.info("penalty_loader_initialized", extra={"source": str(penalties_file)})

def get_base_layer_penalty(self, key: str, default: float = 0.1) -> float:
    """
    Get base layer penalty value.

    Args:
        key: Penalty key (e.g., 'uncalibrated_method')
        default: Default value if not found

    Returns:
        Penalty value
    """
    penalties = self._penalties.get("base_layer_penalties", {})
    penalty_data = penalties.get(key, {})

    if isinstance(penalty_data, dict):
        return penalty_data.get("value", default)
    return default

def get_contextual_layer_penalty(self, key: str, default: float = 0.1) -> float:
    """
    Get contextual layer penalty value.

    Args:
        key: Penalty key (e.g., 'no_compatibility_data_question')
        default: Default value if not found

    Returns:
        Penalty value
    """
    penalties = self._penalties.get("contextual_layer_penalties", {})
    penalty_data = penalties.get(key, {})

    if isinstance(penalty_data, dict):
        return penalty_data.get("value", default)
    return default
```

```
    def get_chain_layer_penalty(self, key: str, default: float = 0.0) -> float:
        """Get chain layer penalty value."""
        penalties = self._penalties.get("chain_layer_penalties", {})
        penalty_data = penalties.get(key, {})

        if isinstance(penalty_data, dict):
            return penalty_data.get("value", default)
        return default

    def get_meta_layer_penalty(self, key: str, default: float = 0.0) -> float:
        """Get meta layer penalty value."""
        penalties = self._penalties.get("meta_layer_penalties", {})
        penalty_data = penalties.get(key, {})

        if isinstance(penalty_data, dict):
            return penalty_data.get("value", default)
        return default


class ThresholdLoader:
    """
    Singleton loader for all quality threshold values.

    Loads from config/quality_thresholds.json
    """

    _instance = None
    _thresholds: Dict[str, Any] = None

    @classmethod
    def get_instance(cls) -> 'ThresholdLoader':
        """Get singleton instance."""
        if cls._instance is None:
            cls._instance = cls()
            cls._instance._load()
        return cls._instance

    def _load(self) -> None:
        """Load thresholds from JSON."""
        thresholds_file = _REPO_ROOT / "config" / "quality_thresholds.json"

        if not thresholds_file.exists():
            raise FileNotFoundError(
                f"Thresholds JSON not found: {thresholds_file}\n"
                f"This file is REQUIRED for ZERO TOLERANCE compliance."
            )

        with open(thresholds_file, 'r') as f:
            self._thresholds = json.load(f)

        logger.info("threshold_loader_initialized", extra={"source":
str(thresholds_file)})

    def get_base_layer_quality_thresholds(self) -> Dict[str, float]:
        """
        Get base layer quality thresholds.

        Returns:
            Dict with keys: excellent, good, acceptable, needs_improvement
        """
        base_quality = self._thresholds.get("base_layer_quality", {})

        return {
            "excellent": base_quality.get("excellent", {}).get("value", 0.8),
            "good": base_quality.get("good", {}).get("value", 0.6),
            "acceptable": base_quality.get("acceptable", {}).get("value", 0.4),
            "needs_improvement": base_quality.get("needs_improvement", {}).get("value",
0.0)
```

```
        }

    def get_final_calibration_quality_thresholds(self) -> Dict[str, float]:
        """
        Get final calibration quality thresholds.

        Returns:
            Dict with keys: excellent, good, acceptable, insufficient
        """
        final_quality = self._thresholds.get("final_calibration_quality", {})

        return {
            "excellent": final_quality.get("excellent", {}).get("value", 0.85),
            "good": final_quality.get("good", {}).get("value", 0.70),
            "acceptable": final_quality.get("acceptable", {}).get("value", 0.55),
            "insufficient": final_quality.get("insufficient", {}).get("value", 0.0)
        }

    def get_executor_threshold(self, executor_name: str, default: float = 0.70) -> float:
        """
        Get validation threshold for specific executor.

        Args:
            executor_name: Executor name (e.g., "D1Q1_Executor")
            default: Default threshold

        Returns:
            Threshold value
        """
        executor_thresholds = self._thresholds.get("executor_specific_thresholds", {})
        executors = executor_thresholds.get("executors", {})

        executor_data = executors.get(executor_name, {})

        if isinstance(executor_data, dict):
            return executor_data.get("value", default)

        # Try default executor threshold
        default_threshold = executor_thresholds.get("default_executor_threshold", {})
        if isinstance(default_threshold, dict):
            return default_threshold.get("value", default)

        return default

===== FILE: src/saaaaaa/core/calibration/congruence_layer.py =====
"""
Congruence Layer (@C) - Full Implementation.

Evaluates method ensemble congruence using three components:
- c_scale: Range compatibility
- c_sem: Semantic overlap (Jaccard index)
- c_fusion: Fusion rule validity

Formula: C_play(G|ctx) = c_scale · c_sem · c_fusion
"""
import logging
from typing import Any

logger = logging.getLogger(__name__)


class CongruenceLayerEvaluator:
    """
    Evaluates congruence of method ensembles.

    Attributes:
        registry: Dictionary mapping method IDs to their metadata
    """
```

```
    def __init__(self, method_registry: dict[str, Any]) -> None:
        """
        Initialize evaluator with method registry.

        Args:
            method_registry: Dict with method metadata (output_range, semantic_tags, etc.)
        """
        self.registry = method_registry
        logger.info("congruence_evaluator_initialized", extra={"num_methods":
len(method_registry)})

    def evaluate(
        self,
        method_ids: list[str],
        subgraph_id: str,
        fusion_rule: str = "weighted_average",
        provided_inputs: list[str] = None
    ) -> float:
        """
        Evaluate congruence of method ensemble.

        Formula: C_play(G|ctx) = c_scale · c_sem · c_fusion

        Args:
            method_ids: List of methods in the subgraph
            subgraph_id: Identifier for this subgraph
            fusion_rule: How outputs are combined (weighted_average, max, min, product)
            provided_inputs: List of actual inputs provided

        Returns:
            C_play ∈ [0.0, 1.0]
        """
        # Edge case: Single method = perfect congruence, but only if method exists
        if len(method_ids) < 2:
            method_id = method_ids[0] if method_ids else None
            if method_id is not None and method_id in self.registry:
                logger.debug("congruence_single_method", extra={"score": 1.0, "method_id":
 method_id})
                return 1.0
            else:
                logger.warning("congruence_single_method_missing", extra={"score": 0.0,
"method_id": method_id})
                return 0.0

        logger.info(
            "congruence_evaluation_start",
            extra={
                "methods": method_ids,
                "subgraph": subgraph_id,
                "fusion": fusion_rule
            }
        )

        # Component 1: Scale congruence (c_scale)
        c_scale = self._compute_scale_congruence(method_ids)
        logger.debug("c_scale_computed", extra={"score": c_scale})

        # Component 2: Semantic congruence (c_sem)
        c_sem = self._compute_semantic_congruence(method_ids)
        logger.debug("c_sem_computed", extra={"score": c_sem})

        # Component 3: Fusion validity (c_fusion)
        c_fusion = self._compute_fusion_validity(
            method_ids, fusion_rule, provided_inputs or []
        )
        logger.debug("c_fusion_computed", extra={"score": c_fusion})
```

```
    # Final score: Product of three components
    C_play = c_scale * c_sem * c_fusion

    logger.info(
        "congruence_computed",
        extra={
            "C_play": C_play,
            "c_scale": c_scale,
            "c_sem": c_sem,
            "c_fusion": c_fusion,
            "subgraph": subgraph_id
        }
    )

    return C_play

def _compute_scale_congruence(self, method_ids: list[str]) -> float:
    """
    Compute c_scale: Range compatibility.

    Scoring:
        1.0: All ranges identical
        0.8: All ranges convertible (within [0,1])
        0.0: Incompatible ranges

    Returns:
        c_scale ∈ {0.0, 0.8, 1.0}
    """
    ranges = []
    for method_id in method_ids:
        if method_id not in self.registry:
            logger.warning("method_not_registered", extra={"method": method_id})
            return 0.0

        method_data = self.registry[method_id]
        output_range = method_data.get("output_range")

        if output_range is None:
            logger.warning("no_output_range", extra={"method": method_id})
            return 0.0

        ranges.append(tuple(output_range))

    # Check if all ranges are identical
    first_range = ranges[0]
    if all(r == first_range for r in ranges):
        logger.debug("ranges_identical", extra={"range": first_range})
        return 1.0

    # Check if all ranges are in [0,1] (convertible)
    all_in_unit = all(r == (0.0, 1.0) for r in ranges)
    if all_in_unit:
        logger.debug("ranges_convertible", extra={"note": "all in [0,1]"})
        return 0.8

    # Incompatible ranges
    logger.warning("ranges_incompatible", extra={"ranges": ranges})
    return 0.0

def _compute_semantic_congruence(self, method_ids: list[str]) -> float:
    """
    Compute c_sem: Semantic overlap (Jaccard index).

    Formula: |intersection| / |union| of semantic tags

    Returns:
        c_sem ∈ [0.0, 1.0]
    """
```

```
    tag_sets = []

    for method_id in method_ids:
        if method_id not in self.registry:
            logger.warning("method_not_registered", extra={"method": method_id})
            return 0.0

        tags = self.registry[method_id].get("semantic_tags", [])
        if not isinstance(tags, (list, set)):
            tags = []
        tag_sets.append(set(tags))

    if not tag_sets:
        return 0.0

    # Compute intersection and union
    intersection = tag_sets[0]
    union = tag_sets[0]

    for tags in tag_sets[1:]:
        intersection = intersection.intersection(tags)
        union = union.union(tags)

    # Jaccard index
    if len(union) == 0:
        logger.warning("no_semantic_tags", extra={"methods": method_ids})
        return 0.0

    jaccard = len(intersection) / len(union)

    logger.debug(
        "semantic_congruence",
        extra={
            "intersection": list(intersection),
            "union_size": len(union),
            "jaccard": jaccard
        }
    )

    return jaccard

def _compute_fusion_validity(
    self,
    method_ids: list[str],
    fusion_rule: str,
    provided_inputs: list[str]
) -> float:
    """
    Compute c_fusion: Fusion rule validity.

    Scoring:
        1.0: Rule valid AND all inputs present
        0.5: Rule valid BUT some inputs missing
        0.0: Rule invalid

    Returns:
        c_fusion ∈ {0.0, 0.5, 1.0}
    """
    # Check if fusion rule is valid
    valid_rules = ["weighted_average", "max", "min", "product", "custom"]
    if fusion_rule not in valid_rules:
        logger.warning(
            "invalid_fusion_rule",
            extra={"rule": fusion_rule, "valid": valid_rules}
        )
        return 0.0

    # Collect all fusion requirements
```

```
        all_requirements = set()
        for method_id in method_ids:
            if method_id not in self.registry:
                return 0.0

            requirements = self.registry[method_id].get("fusion_requirements", [])
            all_requirements.update(requirements)

        # Check if provided inputs cover all requirements
        provided = set(provided_inputs)
        missing = all_requirements - provided

        if not missing:
            # All inputs provided
            logger.debug(
                "fusion_valid",
                extra={"rule": fusion_rule, "all_inputs_present": True}
            )
            return 1.0
        else:
            # Some inputs missing
            logger.warning(
                "fusion_partial",
                extra={
                    "rule": fusion_rule,
                    "missing": list(missing),
                    "provided": list(provided)
                }
            )
            return 0.5

===== FILE: src/saaaaaa/core/calibration/data_structures.py =====
"""
Calibration system data structures.

These dataclasses define the EXACT structure of calibration outputs.
NO fields should be added or removed without updating this spec.

Design Principles:
1. Immutability: All dataclasses are frozen
2. Validation: __post_init__ checks invariants
3. Type Safety: Use type hints everywhere
4. Serializability: Support to_dict() for JSON export
"""
from dataclasses import dataclass, field
from enum import Enum
from typing import Any


# ============================================================================
# EXCEPTIONS
# ============================================================================


class CalibrationConfigError(Exception):
    """
    Raised when calibration configuration is invalid.

    This indicates a config file error that must be fixed before calibration
    can proceed (e.g., invalid fusion weights, missing method declarations).
    """
    pass


# ============================================================================
# ENUMS
# ============================================================================
```

```
class MethodRole(str, Enum):
    """
    Method role in the pipeline.

    These roles determine which fusion weights apply during calibration.
    Per canonic_calibration_methods.md specification.
    """
    INGEST_PDM = "ingest_pdm"      # Ingestion and PDM construction
    STRUCTURE = "structure"        # Text structuring / parsing
    EXTRACT = "extract"            # Pattern extraction
    SCORE_Q = "score_q"            # Question scoring
    AGGREGATE = "aggregate"        # Score aggregation
    REPORT = "report"              # Report formatting
    TRANSFORM = "transform"        # Data transformation / normalization
    META_TOOL = "meta_tool"        # Orchestration / utility methods


class LayerID(str, Enum):
    """
    Exact identifier for each calibration layer.

    These correspond to the 7 layers in the theoretical model.
    """
    BASE = "b"          # @b - Intrinsic quality (COMPLETE)
    UNIT = "u"          # @u - PDT quality
    QUESTION = "q"      # @q - Question compatibility
    DIMENSION = "d"     # @d - Dimension compatibility
    POLICY = "p"        # @p - Policy area compatibility
    CONGRUENCE = "C"    # @C - Ensemble validity
    CHAIN = "chain"     # @chain - Data flow integrity
    META = "m"          # @m - Governance


@dataclass(frozen=True)
class LayerScore:
    """
    Single layer evaluation result.

    This represents the output of evaluating ONE layer for ONE subject.

    Attributes:
        layer: Which layer this score belongs to
        score: Numerical score in [0.0, 1.0]
        components: Breakdown of sub-scores (e.g., for @u: {S, M, I, P})
        rationale: Human-readable explanation of the score
        metadata: Additional debug/audit information

    Example:
        LayerScore(
            layer=LayerID.UNIT,
            score=0.75,
            components={"S": 0.8, "M": 0.7, "I": 0.75, "P": 0.75},
            rationale="Unit quality: robusto (S=0.80, M=0.70, I=0.75, P=0.75)",
            metadata={"aggregation_method": "geometric_mean"}
        )
    """
    layer: LayerID
    score: float  # MUST be in [0.0, 1.0]
    components: dict[str, float] = field(default_factory=dict)
    rationale: str = ""
    metadata: dict[str, Any] = field(default_factory=dict)

    def __post_init__(self):
        """Validate score is in valid range."""
        if not 0.0 <= self.score <= 1.0:
            raise ValueError(
                f"Layer {self.layer.value} score {self.score} out of range [0.0, 1.0]"
```

```
            )

    def to_dict(self) -> dict:
        """Export as dictionary for JSON serialization."""
        return {
            "layer": self.layer.value,
            "score": self.score,
            "components": self.components,
            "rationale": self.rationale,
            "metadata": self.metadata,
        }


@dataclass(frozen=True)
class ContextTuple:
    """
    Execution context for a micro-question: ctx = (Q, D, P, U).

    This is the (Q, D, P, U) tuple that defines WHERE a method is being used.
    The context determines which compatibility scores apply.

    Attributes:
        question_id: e.g., "Q001", "Q031" (from questionnaire monolith)
        dimension: e.g., "DIM01" (canonical code, not "D1")
        policy_area: e.g., "PA01" (canonical code, not "P1")
        unit_quality: Pre-computed U score from PDT analysis, range [0.0, 1.0]

    Example:
        ContextTuple(
            question_id="Q001",
            dimension="DIM01",
            policy_area="PA01",
            unit_quality=0.75
        )
    """
    question_id: str
    dimension: str
    policy_area: str
    unit_quality: float

    def __post_init__(self):
        """Validate canonical notation and ranges."""
        # Validate dimension uses canonical notation
        if not self.dimension.startswith("DIM"):
            raise ValueError(
                f"Dimension must use canonical code (DIM01-DIM06), got {self.dimension}"
            )

        # Validate policy area uses canonical notation
        if not self.policy_area.startswith("PA"):
            raise ValueError(
                f"Policy must use canonical code (PA01-PA10), got {self.policy_area}"
            )

        # Validate question ID format
        if not self.question_id.startswith("Q"):
            raise ValueError(
                f"Question ID must start with 'Q', got {self.question_id}"
            )

        # Validate unit quality range
        if not 0.0 <= self.unit_quality <= 1.0:
            raise ValueError(
                f"Unit quality must be in [0.0, 1.0], got {self.unit_quality}"
            )

    def to_dict(self) -> dict:
        """Export as dictionary."""
```

```
        return {
            "question_id": self.question_id,
            "dimension": self.dimension,
            "policy_area": self.policy_area,
            "unit_quality": self.unit_quality,
        }


@dataclass(frozen=True)
class CalibrationSubject:
    """
    Subject of calibration I = (M, v, Γ, G, ctx).

    This represents ONE method being evaluated in ONE context.

    From theoretical model:
    - M: Method artifact (code)
    - v: Version
    - Γ: Computational graph (how methods connect)
    - G: Interplay subgraph (methods working together)
    - ctx: Context tuple (Q, D, P, U)

    Attributes:
        method_id: e.g., "pattern_extractor_v2"
        method_version: e.g., "v2.1.0"
        graph_config: Hash of the computational graph Γ
        subgraph_id: Identifier for the interplay subgraph G
        context: The (Q, D, P, U) context

    Example:
        CalibrationSubject(
            method_id="pattern_extractor_v2",
            method_version="v2.1.0",
            graph_config="abc123def456",
            subgraph_id="Q001_analyzer_validator",
            context=ContextTuple(...)
        )
    """
    method_id: str
    method_version: str
    graph_config: str
    subgraph_id: str
    context: ContextTuple

    def to_dict(self) -> dict:
        """Export as dictionary."""
        return {
            "method_id": self.method_id,
            "method_version": self.method_version,
            "graph_config": self.graph_config,
            "subgraph_id": self.subgraph_id,
            "context": self.context.to_dict(),
        }


@dataclass(frozen=True)
class CompatibilityMapping:
    """
    Defines how compatible a method is with questions/dimensions/policies.

    This implements the Q_f, D_f, P_f functions from the theoretical model.

    Compatibility Scores (from theoretical model):
        1.0 = Primary (designed specifically for this context)
        0.7 = Secondary (works well, but not optimal)
        0.3 = Compatible (can work, limited effectiveness)
        0.1 = Undeclared (penalty, not validated for this context)
```

```
Example:
    CompatibilityMapping(
        method_id="pattern_extractor_v2",
        questions={"Q001": 1.0, "Q031": 0.7, "Q091": 0.3},
        dimensions={"DIM01": 1.0, "DIM03": 0.7},
        policies={"PA01": 1.0, "PA10": 0.7}
    )
"""
method_id: str
questions: dict[str, float]   # question_id -> score ∈ {1.0, 0.7, 0.3, 0.1}
dimensions: dict[str, float]  # dimension_code -> score
policies: dict[str, float]    # policy_code -> score

def get_question_score(self, question_id: str) -> float:
    """
    Get compatibility score for a question.

    Returns 0.1 (penalty) if question not declared.
    """
    return self.questions.get(question_id, 0.1)

def get_dimension_score(self, dimension: str) -> float:
    """
    Get compatibility score for a dimension.

    Returns 0.1 (penalty) if dimension not declared.
    """
    return self.dimensions.get(dimension, 0.1)

def get_policy_score(self, policy: str) -> float:
    """
    Get compatibility score for a policy area.

    Returns 0.1 (penalty) if policy not declared.
    """
    return self.policies.get(policy, 0.1)

def check_anti_universality(self, threshold: float = 0.9) -> bool:
    """
    Check Anti-Universality Theorem compliance.

    The theorem states: NO method can have average compatibility ≥ 0.9
    across ALL questions, dimensions, AND policies simultaneously.

    Returns:
        True if compliant (method is NOT universal)
        False if violation detected
    """
    if not self.questions or not self.dimensions or not self.policies:
        return True  # Incomplete mapping, cannot be universal

    avg_q = sum(self.questions.values()) / len(self.questions)
    avg_d = sum(self.dimensions.values()) / len(self.dimensions)
    avg_p = sum(self.policies.values()) / len(self.policies)

    is_universal = (avg_q >= threshold and
                    avg_d >= threshold and
                    avg_p >= threshold)

    return not is_universal

def to_dict(self) -> dict:
    """Export as dictionary."""
    return {
        "method_id": self.method_id,
        "questions": self.questions,
        "dimensions": self.dimensions,
        "policies": self.policies,
```

```
        }


@dataclass(frozen=True)
class InteractionTerm:
    """
    Represents a synergy between two layers in Choquet aggregation.

    Formula: a_ℓk · min(x_ℓ, x_k)

    This captures the "weakest link" principle: the contribution of the
    interaction is limited by whichever layer scored lower.

    Standard Interactions (from theoretical model):
        (@u, @chain): weight=0.15, "Plan quality only matters with sound wiring"
        (@chain, @C): weight=0.12, "Ensemble validity requires chain integrity"
        (@q, @d): weight=0.08, "Question-dimension alignment synergy"
        (@d, @p): weight=0.05, "Dimension-policy coherence synergy"

    Example:
        InteractionTerm(
            layer_1=LayerID.UNIT,
            layer_2=LayerID.CHAIN,
            weight=0.15,
            rationale="Plan quality only matters with sound wiring"
        )
    """
    layer_1: LayerID
    layer_2: LayerID
    weight: float  # a_ℓk coefficient
    rationale: str  # Why this interaction exists

    def compute(self, scores: dict[LayerID, float]) -> float:
        """
        Compute interaction contribution.

        Formula: a_ℓk · min(x_ℓ, x_k)

        Args:
            scores: Dictionary mapping LayerID to score

        Returns:
            Interaction contribution (can be 0 if layer missing)
        """
        score_1 = scores.get(self.layer_1, 0.0)
        score_2 = scores.get(self.layer_2, 0.0)
        return self.weight * min(score_1, score_2)

    def to_dict(self) -> dict:
        """Export as dictionary."""
        return {
            "layer_1": self.layer_1.value,
            "layer_2": self.layer_2.value,
            "weight": self.weight,
            "rationale": self.rationale,
        }


@dataclass(frozen=True)
class CalibrationResult:
    """
    Complete calibration output for a subject I.

    This is the FINAL result of the calibration pipeline.

    Formula: Cal(I) = Σ a_ℓ·x_ℓ + Σ a_ℓk·min(x_ℓ, x_k)

    Attributes:
```

```
    subject: The calibration subject I = (M, v, Γ, G, ctx)
    layer_scores: Individual scores for each layer
    linear_contribution: Σ a_ℓ · x_ℓ
    interaction_contribution: Σ a_ℓk · min(x_ℓ, x_k)
    final_score: Cal(I) = linear + interaction ∈ [0.0, 1.0]
    computation_metadata: Timestamps, hashes, config_hash, etc.

Example:
    CalibrationResult(
        subject=CalibrationSubject(...),
        layer_scores={
            LayerID.BASE: LayerScore(..., score=0.9),
            LayerID.UNIT: LayerScore(..., score=0.75),
            ...
        },
        linear_contribution=0.65,
        interaction_contribution=0.15,
        final_score=0.80,
        computation_metadata={
            "config_hash": "abc123",
            "timestamp": "2025-11-11T10:30:00Z"
        }
    )
"""
subject: CalibrationSubject
layer_scores: dict[LayerID, LayerScore]
linear_contribution: float
interaction_contribution: float
final_score: float
computation_metadata: dict[str, Any] = field(default_factory=dict)

def __post_init__(self):
    """Validate calibration result integrity."""
    # Validate final score range
    if not 0.0 <= self.final_score <= 1.0:
        raise ValueError(
            f"Final calibration score {self.final_score} out of range [0.0, 1.0]"
        )

    # Verify linear + interaction = final (within numerical tolerance)
    computed = self.linear_contribution + self.interaction_contribution
    if abs(computed - self.final_score) > 1e-6:
        raise ValueError(
            f"Final score {self.final_score} != "
            f"linear {self.linear_contribution} + "
            f"interaction {self.interaction_contribution} = {computed}"
        )

    # Verify all layer scores are in valid range
    for layer_id, layer_score in self.layer_scores.items():
        if not 0.0 <= layer_score.score <= 1.0:
            raise ValueError(
                f"Layer {layer_id.value} score {layer_score.score} out of range"
            )

def to_certificate_dict(self) -> dict:
    """
    Export as a calibration certificate for auditing.

    This is the format that gets saved to audit logs and can be
    used to reproduce the calibration result.
    """
    return {
        "certificate_version": "1.0",
        "method": {
            "id": self.subject.method_id,
            "version": self.subject.method_version,
            "graph_config": self.subject.graph_config,
```

```
            "subgraph_id": self.subject.subgraph_id,
        },
        "context": self.subject.context.to_dict(),
        "layer_scores": {
            layer_id.value: layer_score.to_dict()
            for layer_id, layer_score in self.layer_scores.items()
        },
        "aggregation": {
            "linear_contribution": self.linear_contribution,
            "interaction_contribution": self.interaction_contribution,
            "final_score": self.final_score,
            "formula": "Cal(I) = Σ a_ℓ·x_ℓ + Σ a_ℓk·min(x_ℓ, x_k)",
        },
        "metadata": self.computation_metadata,
    }

def to_dict(self) -> dict:
    """Export as dictionary."""
    return self.to_certificate_dict()


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


@dataclass
class ComputationGraph:
    """
    Represents a computational graph for method execution.

    This is a simplified representation of how methods connect in a pipeline.

    Attributes:
        nodes: List of node identifiers
        edges: List of (source, target) edges
        node_signatures: Dict mapping node_id to signature info
    """
    nodes: list[str]
    edges: list[tuple[str, str]]
    node_signatures: dict[str, dict[str, Any]] = field(default_factory=dict)

    def validate_dag(self) -> bool:
        """
        Validate that the graph is a Directed Acyclic Graph (DAG).

        Returns:
            True if graph is a valid DAG, False if cycles detected
        """
        # Simple cycle detection using DFS
        visited = set()
        rec_stack = set()

        def has_cycle(node: str) -> bool:
            visited.add(node)
            rec_stack.add(node)

            # Get neighbors
            neighbors = [target for source, target in self.edges if source == node]

            for neighbor in neighbors:
                if neighbor not in visited:
                    if has_cycle(neighbor):
                        return True
                elif neighbor in rec_stack:
                    return True

            rec_stack.remove(node)
```

```
            return False

        # Check each node
        for node in self.nodes:
            if node not in visited:
                if has_cycle(node):
                    return False

        return True


@dataclass
class EvidenceStore:
    """
    Evidence collected during method execution for calibration.

    Attributes:
        pdt_structure: PDT (Policy Document Tree) metrics
        runtime_metrics: Execution time and resource usage
        validation_results: Any validation checks performed
    """
    pdt_structure: dict[str, Any] = field(default_factory=dict)
    runtime_metrics: dict[str, Any] = field(default_factory=dict)
    validation_results: dict[str, Any] = field(default_factory=dict)


# Type alias for interplay subgraph (simplified representation)
InterplaySubgraph = dict[str, Any]


@dataclass(frozen=True)
class CalibrationCertificate:
    """
    Complete calibration certificate with audit trail.

    This is the output of the calibration engine, containing all information
    needed to verify and reproduce the calibration.

    Attributes:
        instance_id: Unique identifier for this calibration instance
        method_id: Canonical method identifier
        node_id: Node identifier in computation graph
        context: Execution context (Q, D, P, U)
        intrinsic_score: Base layer (@b) score
        layer_scores: All layer scores
        calibrated_score: Final calibrated score
        fusion_formula: Fusion computation details
        parameter_provenance: Where parameters came from
        evidence_trail: Evidence used for calibration
        config_hash: Hash of configuration files
        graph_hash: Hash of computation graph
        timestamp: When calibration was performed
        validator_version: Version of calibration system
    """
    instance_id: str
    method_id: str
    node_id: str
    context: ContextTuple
    intrinsic_score: float
    layer_scores: dict[str, float]
    calibrated_score: float
    fusion_formula: dict[str, Any]
    parameter_provenance: dict[str, Any]
    evidence_trail: dict[str, Any]
    config_hash: str
    graph_hash: str
    timestamp: str
    validator_version: str
```

```
# ============================================================================
# CONSTANTS
# ============================================================================


# Required layers for each method role
# Per canonic_calibration_methods.md specification
REQUIRED_LAYERS: dict[MethodRole, set[LayerID]] = {
    MethodRole.INGEST_PDM: {
        LayerID.BASE,
        LayerID.UNIT,
        LayerID.CHAIN,
        LayerID.META,
    },
    MethodRole.STRUCTURE: {
        LayerID.BASE,
        LayerID.CHAIN,
        LayerID.META,
    },
    MethodRole.EXTRACT: {
        LayerID.BASE,
        LayerID.QUESTION,
        LayerID.DIMENSION,
        LayerID.CHAIN,
        LayerID.META,
    },
    MethodRole.SCORE_Q: {
        LayerID.BASE,
        LayerID.QUESTION,
        LayerID.DIMENSION,
        LayerID.POLICY,
        LayerID.CHAIN,
        LayerID.META,
    },
    MethodRole.AGGREGATE: {
        LayerID.BASE,
        LayerID.DIMENSION,
        LayerID.POLICY,
        LayerID.CONGRUENCE,
        LayerID.CHAIN,
        LayerID.META,
    },
    MethodRole.REPORT: {
        LayerID.BASE,
        LayerID.CHAIN,
        LayerID.META,
    },
    MethodRole.TRANSFORM: {
        LayerID.BASE,
        LayerID.CHAIN,
        LayerID.META,
    },
    MethodRole.META_TOOL: {
        LayerID.BASE,
        LayerID.META,
    },
}

===== FILE: src/saaaaaa/core/calibration/decorators.py =====
"""Calibration Decorators.

This module provides decorators to enforce the central calibration system.
"""

from functools import wraps
from dataclasses import dataclass
```

```
from typing import Any, Dict, Optional
import logging

from saaaaaa import get_calibration_orchestrator, get_parameter_loader

logger = logging.getLogger(__name__)

class CalibrationError(Exception):
    """Raised when calibration fails."""
    pass

@dataclass
class CalibratedResult:
    """Result wrapper for calibrated methods."""
    value: Any
    calibration_score: float
    layer_scores: Dict[str, float]
    metadata: Dict[str, Any]

def calibrated_method(method_id: str):
    """
    OBLIGATORY: Decorator that FORCES anchoring to the central system.

    USAGE:
        @calibrated_method("module.Class.method")
        def my_method(self, data):
            # Your code here
            return result

    The decorator:
    1. Loads parameters from JSON
    2. Executes the method
    3. Calibrates the result
    4. Validates and returns
    """
    def decorator(func):
        @wraps(func)
        def wrapper(self, *args, **kwargs):
            # 1. GET central system
            orchestrator = get_calibration_orchestrator()
            param_loader = get_parameter_loader()

            # 2. LOAD parameters
            params = param_loader.get(method_id)

            # Merge params into kwargs if they don't exist
            # This allows overriding from caller but defaults to JSON
            for k, v in params.items():
                if k not in kwargs:
                    kwargs[k] = v

            # 3. EXECUTE original method
            try:
                raw_result = func(self, *args, **kwargs)
            except Exception as e:
                logger.error(f"Method {method_id} execution failed: {e}")
                raise

            # 4. CALIBRATE result
            context = {
                "method_id": method_id,
                "args": args,
                "kwargs": kwargs,
                "instance": self,
                "raw_result": raw_result
            }

            try:
```

```
            calibration = orchestrator.calibrate(method_id, context)
        except Exception as e:
            logger.error(f"Calibration failed for {method_id}: {e}")
            # Fail safe or raise? Request implies strictness.
            raise CalibrationError(f"Calibration system failure: {e}")

        # 5. VALIDATE
        threshold = params.get("validation_threshold", 0.7)

        if calibration.final_score < threshold:
            logger.warning(
                f"Method {method_id} failed calibration: "
                f"score {calibration.final_score:.3f} < threshold {threshold}"
            )
            # Depending on strictness, we might raise or return a failure object.
            # The prompt example raised CalibrationError.
            raise CalibrationError(
                f"Method {method_id} failed calibration: "
                f"score {calibration.final_score:.3f} < threshold {threshold}"
            )

        # 6. RETURN result with metadata
        # If the method returns a complex object, we might want to attach metadata to
it
        # Or return a wrapper. The prompt example returns CalibratedResult.
        # However, this changes the return type signature.
        # If the original code expects the raw result, this might break things.
        # BUT the prompt explicitly says: "return CalibratedResult"
        # So I will follow the prompt.

        return CalibratedResult(
            value=raw_result,
            calibration_score=calibration.final_score,
            layer_scores=calibration.layer_scores,
            metadata=calibration.metadata
        )

    return wrapper
  return decorator

# Need to import dataclasses
from dataclasses import dataclass

===== FILE: src/saaaaaa/core/calibration/engine.py =====
"""
Three-Pillar Calibration System - Main Calibration Engine

This module implements the main calibrate() function and fusion operator
as specified in the SUPERPROMPT Three-Pillar Calibration System.

Spec compliance: Section 5 (Fusion Operator), Section 6 (Runtime Engine)
SIN_CARRETA Compliance: Pure fusion operator, fail-loudly on misconfiguration
"""

import hashlib
import json
from datetime import datetime, timezone
from pathlib import Path
from typing import Any

from .data_structures import (
  REQUIRED_LAYERS,
  CalibrationCertificate,
  CalibrationConfigError,
  CalibrationSubject,
  ComputationGraph,
  ContextTuple,
  EvidenceStore,
```

```
    LayerID,
    MethodRole,
)
from .layer_computers import (
    compute_base_layer,
    compute_chain_layer,
    compute_dimension_layer,
    compute_interplay_layer,
    compute_meta_layer,
    compute_policy_layer,
    compute_question_layer,
    compute_unit_layer,
)


class CalibrationEngine:
    """
    Main calibration engine implementing the three-pillar system.

    Spec compliance: Section 7 (Runtime Engine & Certificate)
    """

    def __init__(self, config_dir: str = None, monolith_path: str = None, catalog_path:
str = None) -> None:
        """
        Initialize calibration engine and load configs.

        SIN_CARRETA: Validates fusion weights at load time to fail fast.
        Three-Pillar System: Loads from intrinsic, contextual, and fusion configs.

        Args:
            config_dir: Path to config directory (defaults to ../config)
            monolith_path: Path to questionnaire_monolith.json (defaults to
../data/questionnaire_monolith.json)
            catalog_path: Path to canonical_method_catalog.json (defaults to
../config/canonical_method_catalog.json)

        Raises:
            CalibrationConfigError: If fusion weights violate constraints
        """
        if config_dir is None:
            config_dir = Path(__file__).parent.parent / "config"
        else:
            config_dir = Path(config_dir)

        self.config_dir = config_dir
        self.intrinsic_config = self._load_json(config_dir / "intrinsic_calibration.json")
        self.contextual_config = self._load_json(config_dir /
"contextual_parametrization.json")
        self.fusion_config = self._load_json(config_dir / "fusion_specification.json")

        # Load canonical method catalog for role determination
        if catalog_path is None:
            catalog_path = config_dir / "canonical_method_catalog.json"
        else:
            catalog_path = Path(catalog_path)
        self.catalog = self._load_json(catalog_path)
        self._build_method_index()

        # SIN_CARRETA: Validate fusion weights at load time
        self._validate_fusion_weights()

        # Load questionnaire monolith using canonical loader
        # This ensures hash verification and immutability
        from saaaaaa.core.orchestrator.factory import load_questionnaire

        if monolith_path is None:
            # Use default path from factory
```

```
        canonical_q = load_questionnaire()
    else:
        # Use specified path
        canonical_q = load_questionnaire(Path(monolith_path))

    # Convert to dict for backward compatibility with calibration system
    # (CalibrationEngine expects dict, not CanonicalQuestionnaire)
    self.monolith = dict(canonical_q.data)
    self._questionnaire_hash = canonical_q.sha256  # Store for verification

    # Compute config hash
    self.config_hash = self._compute_config_hash()

@staticmethod
def _load_json(path: Path) -> dict[str, Any]:
    """Load JSON file"""
    with open(path) as f:
        return json.load(f)

def _build_method_index(self) -> None:
    """
    Build index of methods from canonical catalog for fast role lookup.

    Three-Pillar System: Uses canonical_method_catalog.json as single source.
    """
    self.method_index = {}

    for _layer_name, methods in self.catalog.get("layers", {}).items():
        for method_info in methods:
            canonical_name = method_info.get("canonical_name", "")
            method_name = method_info.get("method_name", "")
            class_name = method_info.get("class_name", "")
            layer = method_info.get("layer", "unknown")

            # Store method info with multiple lookup keys
            for key in [canonical_name, method_name, f"{class_name}.{method_name}"]:
                if key:
                    self.method_index[key] = {
                        "canonical_name": canonical_name,
                        "method_name": method_name,
                        "class_name": class_name,
                        "layer": layer,
                        "metadata": method_info
                    }

def _validate_fusion_weights(self) -> None:
    """
    Validate fusion weight constraints at config load time.

    Per canonic_calibration_methods.md specification.

    Constraints:
    1. All weights must be non-negative: a_ℓ ≥ 0, a_ℓk ≥ 0
    2. Total weight sum MUST equal 1: Σ(a_ℓ) + Σ(a_ℓk) = 1 (tolerance 1e-9)

    Raises:
        CalibrationConfigError: If any weight constraint is violated
    """
    role_params_dict = self.fusion_config.get("role_fusion_parameters", {})
    TOLERANCE = 1e-9

    for role_name, role_params in role_params_dict.items():
        linear_weights = role_params.get("linear_weights", {})
        interaction_weights = role_params.get("interaction_weights", {})

        # Constraint 1: Non-negativity
        for layer, weight in linear_weights.items():
            if weight < 0:
```

```
                raise CalibrationConfigError(
                    f"Negative weight for role={role_name}, layer={layer}: "
                    f"weight={weight}. All weights must be ≥ 0."
                )

        for pair, weight in interaction_weights.items():
            if weight < 0:
                raise CalibrationConfigError(
                    f"Negative interaction weight for role={role_name}, pair={pair}: "
                    f"weight={weight}. All weights must be ≥ 0."
                )

        # Constraint 2: Must sum to exactly 1.0
        total_weight = sum(linear_weights.values()) +
sum(interaction_weights.values())
        if abs(total_weight - 1.0) > TOLERANCE:
            raise CalibrationConfigError(
                f"Weight sum must equal 1.0 for role={role_name}: "
                f"total_weight={total_weight:.15f} (deviation: {abs(total_weight -
1.0):.15f}). "
                f"Constraint: Σ(a_ℓ) + Σ(a_ℓk) = 1.0 (tolerance {TOLERANCE})."
            )

    def _compute_config_hash(self) -> str:
        """
        Compute SHA256 hash of all config files.

        Spec compliance: Section 7 (audit_trail.config_hash)
        """
        hasher = hashlib.sha256()

        # Hash all three pillar configs in sorted order
        for config in sorted([
            json.dumps(self.intrinsic_config, sort_keys=True),
            json.dumps(self.contextual_config, sort_keys=True),
            json.dumps(self.fusion_config, sort_keys=True),
        ]):
            hasher.update(config.encode('utf-8'))

        return f"sha256:{hasher.hexdigest()}"

    @staticmethod
    def _compute_graph_hash(graph: ComputationGraph) -> str:
        """
        Compute SHA256 hash of computation graph.

        Spec compliance: Section 7 (audit_trail.graph_hash)
        """
        hasher = hashlib.sha256()

        # Hash nodes and edges
        graph_repr = json.dumps({
            "nodes": sorted(graph.nodes),
            "edges": sorted([list(e) for e in graph.edges])
        }, sort_keys=True)

        hasher.update(graph_repr.encode('utf-8'))
        return f"sha256:{hasher.hexdigest()}"

    def _determine_role(self, method_id: str) -> MethodRole:
        """
        Determine method role from method ID using canonical catalog metadata.

        Three-Pillar System: Uses canonical_method_catalog.json for role inference.
        Mapping from catalog layer + method patterns to MethodRole enum.

        Args:
            method_id: Method identifier (canonical_name, method_name, or Class.method
```

```
format)

        Returns:
            MethodRole enum value

        Raises:
            CalibrationConfigError: If method not found in catalog or role cannot be
determined
        """
        # Look up method in catalog index
        method_info = self.method_index.get(method_id)

        if not method_info:
            # Try fallback patterns
            for key, info in self.method_index.items():
                if method_id in key or key in method_id:
                    method_info = info
                    break

        if not method_info:
            # Cannot calibrate unknown methods - fail loudly
            raise CalibrationConfigError(
                f"Method '{method_id}' not found in canonical_method_catalog.json. "
                f"Cannot determine role for calibration. "
                f"All calibrated methods must be registered in catalog.\n"
                f"To resolve:\n"
                f"  1. Add method to config/canonical_method_catalog.json with proper
metadata\n"
                f"  2. Run scripts/rigorous_calibration_triage.py to generate intrinsic
calibration\n"
                f"  3. Ensure method has correct layer, role, and signature information"
            )

        # Determine role from layer + method name patterns (per
canonic_calibration_methods.md)
        layer = method_info.get("layer", "unknown")
        method_name = method_info.get("method_name", "").lower()

        # Role mapping based on layer and method semantics
        # Per L_* specification in canonic_calibration_methods.md
        if layer == "ingestion" or "ingest" in method_name or "pdm" in method_name:
            return MethodRole.INGEST_PDM
        elif "structure" in method_name or "parse" in method_name:
            return MethodRole.STRUCTURE
        elif "extract" in method_name:
            return MethodRole.EXTRACT
        elif "score" in method_name or "question" in method_name or layer == "analyzer":
            return MethodRole.SCORE_Q
        elif "aggregate" in method_name or "combine" in method_name:
            return MethodRole.AGGREGATE
        elif "report" in method_name or "format" in method_name:
            return MethodRole.REPORT
        elif "transform" in method_name or "normalize" in method_name or "convert" in
method_name:
            return MethodRole.TRANSFORM
        else:
            # Default to META_TOOL for utility/orchestrator methods
            return MethodRole.META_TOOL

    def _detect_interplay(self, graph: ComputationGraph, node_id: str) -> Any | None:
        """
        Detect interplay patterns from computation graph.

        Three-Pillar System: Interplays are DECLARED in config, not auto-detected.

        Per canonic_calibration_methods.md Section 1.3:
        - "An interplay G is valid only if all nodes share a single declared target
output"
```

```
    - "A fusion rule is declared in config"
    - "Do not infer ensembles implicitly"

    This method checks if the node participates in any declared interplay
    from the contextual config.

    Args:
        graph: Computation graph
        node_id: Node identifier

    Returns:
        Interplay subgraph if node participates in one, None otherwise
    """
    # Per specification: interplays are declared in contextual config, not inferred
    # Check contextual_parametrization.json for declared interplays
    interplay_defs = self.contextual_config.get("interplay_definitions", {})

    for interplay_id, interplay_spec in interplay_defs.items():
        # Check if node_id is in this interplay's participant list
        participants = interplay_spec.get("participants", [])
        if node_id in participants:
            # Node participates in this declared interplay
            # Return interplay specification
            return {
                "interplay_id": interplay_id,
                "participants": participants,
                "target_output": interplay_spec.get("target_output"),
                "fusion_rule": interplay_spec.get("fusion_rule"),
                "declared": True
            }

    # Node does not participate in any declared interplay
    # This is normal - most nodes don't participate in interplays
    return None

def _compute_layer_scores(
    self,
    subject: CalibrationSubject,
    evidence: EvidenceStore
) -> dict[str, float]:
    """
    Compute all layer scores for calibration subject.

    Spec compliance: Section 3 (all layers)
    """
    ctx = subject.context
    scores = {}

    # @b: Base layer (always required)
    scores[LayerID.BASE.value] = compute_base_layer(
        subject.method_id, self.intrinsic_config
    )

    # @chain: Chain compatibility (always required for non-META roles)
    scores[LayerID.CHAIN.value] = compute_chain_layer(
        subject.node_id, subject.graph, self.contextual_config
    )

    # @u: Unit-of-analysis
    if subject.role:
        scores[LayerID.UNIT.value] = compute_unit_layer(
            subject.method_id, subject.role, ctx.unit_quality, self.contextual_config
        )

    # @q: Question compatibility
    scores[LayerID.QUESTION.value] = compute_question_layer(
        subject.method_id, ctx.question_id, self.monolith, self.contextual_config
    )
```

```
        # @d: Dimension compatibility
        scores[LayerID.DIMENSION.value] = compute_dimension_layer(
            subject.method_id, ctx.dimension, self.contextual_config
        )

        # @p: Policy compatibility
        scores[LayerID.POLICY.value] = compute_policy_layer(
            subject.method_id, ctx.policy_area, self.contextual_config
        )

        # @C: Interplay congruence
        scores[LayerID.CONGRUENCE.value] = compute_interplay_layer(
            subject.interplay, self.contextual_config
        )

        # @m: Meta/governance
        meta_evidence = {
            "formula_export_valid": True,
            "trace_complete": True,
            "logs_conform_schema": True,
            "version_tagged": True,
            "config_hash_matches": True,
            "signature_valid": True,
            "runtime_ms": evidence.runtime_metrics.get("runtime_ms", 100)
        }
        scores[LayerID.META.value] = compute_meta_layer(
            meta_evidence, self.contextual_config
        )

        return scores

    def _apply_fusion(
        self,
        role: MethodRole,
        layer_scores: dict[str, float]
    ) -> tuple[float, dict[str, Any]]:
        """
        Apply pure fusion operator to combine layer scores.

        Spec compliance: Section 5 (Fusion Operator)
        SIN_CARRETA Compliance: Pure mathematical formula, no clamping/normalization

        Formula: Cal(I) = Σ(a_ℓ · x_ℓ) + Σ(a_ℓk · min(x_ℓ, x_k))

        Weight constraints (enforced at load time):
        - All weights a_ℓ, a_ℓk ≥ 0
        - Σ(a_ℓ) + Σ(a_ℓk) ≤ 1 (ensures boundedness)

        Returns:
            (calibrated_score, fusion_details)

        Raises:
            CalibrationConfigError: If score violates [0,1] bounds (weight
misconfiguration)
        """
        role_params = self.fusion_config["role_fusion_parameters"].get(
            role.value,
            self.fusion_config["default_fallback"]
        )

        linear_weights = role_params["linear_weights"]
        interaction_weights = role_params.get("interaction_weights", {})

        # Compute linear terms
        linear_sum = 0.0
        linear_trace = []
```

```
        for layer_key, weight in linear_weights.items():
            if layer_key in layer_scores:
                contribution = weight * layer_scores[layer_key]
                linear_sum += contribution
                linear_trace.append({
                    "layer": layer_key,
                    "weight": weight,
                    "score": layer_scores[layer_key],
                    "contribution": contribution
                })

        # Compute interaction terms
        interaction_sum = 0.0
        interaction_trace = []

        for pair_key, weight in interaction_weights.items():
            # Parse "(layer1, layer2)" format
            pair_str = pair_key.strip("()")
            layer1, layer2 = [l.strip() for l in pair_str.split(",")]

            if layer1 in layer_scores and layer2 in layer_scores:
                min_score = min(layer_scores[layer1], layer_scores[layer2])
                contribution = weight * min_score
                interaction_sum += contribution
                interaction_trace.append({
                    "pair": pair_key,
                    "weight": weight,
                    "layer1_score": layer_scores[layer1],
                    "layer2_score": layer_scores[layer2],
                    "min_score": min_score,
                    "contribution": contribution
                })

        # Total calibrated score (PURE FUSION - no clamping or normalization)
        calibrated_score = linear_sum + interaction_sum

        # SIN_CARRETA: Fail loudly on weight misconfiguration
        # NEVER clamp or normalize - that would hide misconfiguration
        if calibrated_score < 0.0 or calibrated_score > 1.0:
            total_weight = sum(linear_weights.values()) +
sum(interaction_weights.values())
            raise CalibrationConfigError(
                f"Fusion weights misconfigured for role {role.value}: "
                f"total_weight={total_weight:.6f} produced
calibrated_score={calibrated_score:.6f}. "
                f"Score must be in [0,1]. Weight constraints violated. "
                f"Check fusion_specification.json and ensure Σ(a_ℓ) + Σ(a_ℓk) ≤ 1."
            )

        fusion_details = {
            "symbolic": "Σ(a_ℓ·x_ℓ) + Σ(a_ℓk·min(x_ℓ,x_k))",
            "linear_terms": linear_trace,
            "interaction_terms": interaction_trace,
            "linear_sum": linear_sum,
            "interaction_sum": interaction_sum,
            "total": calibrated_score
        }

        return calibrated_score, fusion_details

    def calibrate(
        self,
        method_id: str,
        node_id: str,
        graph: ComputationGraph,
        context: ContextTuple,
        evidence_store: EvidenceStore
    ) -> CalibrationCertificate:
```

```
"""
Main calibration function.

Spec compliance: Section 7 (Runtime Engine)

Args:
    method_id: Canonical method ID
    node_id: Node identifier in graph
    graph: Computation graph
    context: Execution context
    evidence_store: Evidence for calibration

Returns:
    CalibrationCertificate with complete audit trail

Raises:
    ValueError: If validation fails
"""
# Validate graph is DAG
if not graph.validate_dag():
    raise ValueError("Graph contains cycles - must be DAG")

# Determine role
role = self._determine_role(method_id)

# SIN_CARRETA: Detect interplay from graph (fail if not implemented)
interplay = self._detect_interplay(graph, node_id)

# Create calibration subject
subject = CalibrationSubject(
    method_id=method_id,
    node_id=node_id,
    graph=graph,
    interplay=interplay,
    context=context,
    role=role
)

# Validate layer completeness
required = REQUIRED_LAYERS.get(role, set())

# Compute layer scores
layer_scores = self._compute_layer_scores(subject, evidence_store)

# Check all required layers are present
missing_layers = [layer for layer in required if layer.value not in layer_scores]
if missing_layers:
    raise ValueError(
        f"Missing required layers for role {role.value}: "
        f"{[layer.value for layer in missing_layers]}"
    )

# Apply fusion
calibrated_score, fusion_details = self._apply_fusion(role, layer_scores)

# Build parameter provenance
role_params = self.fusion_config["role_fusion_parameters"].get(
    role.value,
    self.fusion_config["default_fallback"]
)

parameter_provenance = {
    "fusion_weights": {
        "source": "fusion_specification.json",
        "role": role.value,
        "linear_weights": role_params["linear_weights"],
        "interaction_weights": role_params.get("interaction_weights", {})
    },
```

```
            "intrinsic_calibration": {
                "source": "intrinsic_calibration.json",
                "method_id": method_id
            }
        }

        # Build evidence trail
        evidence_trail = {
            "pdt_metrics": evidence_store.pdt_structure,
            "runtime_metrics": evidence_store.runtime_metrics,
            "layer_computations": layer_scores
        }

        # Create certificate
        certificate = CalibrationCertificate(
            instance_id=f"{method_id}@{node_id}",
            method_id=method_id,
            node_id=node_id,
            context=context,
            intrinsic_score=layer_scores.get(LayerID.BASE.value, 0.0),
            layer_scores=layer_scores,
            calibrated_score=calibrated_score,
            fusion_formula=fusion_details,
            parameter_provenance=parameter_provenance,
            evidence_trail=evidence_trail,
            config_hash=self.config_hash,
            graph_hash=self._compute_graph_hash(graph),
            timestamp=datetime.now(timezone.utc).isoformat().replace('+00:00', 'Z'),
            validator_version="1.0.0"
        )

        return certificate


# Convenience function
def calibrate(
    method_id: str,
    node_id: str,
    graph: ComputationGraph,
    context: ContextTuple,
    evidence_store: EvidenceStore,
    config_dir: str | None = None,
    monolith_path: str | None = None
) -> CalibrationCertificate:
    """
    Calibrate a method instance.

    Spec compliance: Section 7

    This is the single authoritative calibration entry point.
    """
    engine = CalibrationEngine(config_dir=config_dir, monolith_path=monolith_path)
    return engine.calibrate(method_id, node_id, graph, context, evidence_store)

===== FILE: src/saaaaaa/core/calibration/intrinsic_loader.py =====
"""Intrinsic Calibration Loader.

Singleton loader for intrinsic_calibration.json (System 2).
"""

import json
import logging
from pathlib import Path
from typing import Any, Dict, Optional

logger = logging.getLogger(__name__)

class IntrinsicCalibrationLoader:
```

```
    _instance = None
    _data: Dict[str, Any] = {}
    _loaded = False

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(IntrinsicCalibrationLoader, cls).__new__(cls)
        return cls._instance

    def load(self, config_path: str = "config/intrinsic_calibration.json") -> None:
        """Load intrinsic calibration data from JSON."""
        if self._loaded:
            return

        path = Path(config_path)
        if not path.exists():
            # Try finding it relative to repo root if running from src
            repo_root = Path(__file__).parent.parent.parent.parent.parent
            path = repo_root / config_path

        if not path.exists():
             logger.warning(f"Intrinsic calibration file not found at {path}. Using empty
config.")
             self._data = {}
             self._loaded = True
             return

        try:
            with open(path, "r") as f:
                self._data = json.load(f)
            self._loaded = True
            logger.info(f"Loaded intrinsic calibration from {path}")
        except Exception as e:
            logger.error(f"Failed to load intrinsic calibration: {e}")
            self._data = {}

    def get_intrinsic_score(self, method_id: str) -> float:
        """Get intrinsic score (@b) for a method."""
        if not self._loaded:
            self.load()

        method_data = self._data.get(method_id)
        if method_data:
            return method_data.get("intrinsic_score", 0.5)
        return 0.5

    def get_metadata(self, method_id: str) -> Optional[Dict[str, Any]]:
        """Get full metadata for a method."""
        if not self._loaded:
            self.load()
        return self._data.get(method_id)

===== FILE: src/saaaaaa/core/calibration/layer_computers.py =====
"""
Three-Pillar Calibration System - Layer Computation Functions

This module implements the 8 layer score computation functions as specified
in the SUPERPROMPT Three-Pillar Calibration System.

Spec compliance: Section 3 (Layer Architecture)
"""

import math
from typing import Any

from .data_structures import (
    CalibrationConfigError,
    ComputationGraph,
```

```
    InterplaySubgraph,
    MethodRole,
)


def compute_base_layer(method_id: str, intrinsic_config: dict[str, Any]) -> float:
    """
    Compute base layer score (@b): Intrinsic quality

    Spec compliance: Section 3.1
    Formula: x_@b = w_th · b_theory + w_imp · b_impl + w_dep · b_deploy

    Args:
        method_id: Canonical method ID
        intrinsic_config: Loaded intrinsic_calibration.json

    Returns:
        Score in [0,1]

    Raises:
        ValueError: If method not found or scores invalid
    """
    if method_id not in intrinsic_config.get("methods", {}):
        raise ValueError(f"Method {method_id} not found in intrinsic_calibration.json")

    method_data = intrinsic_config["methods"][method_id]
    weights = intrinsic_config["_base_weights"]

    b_theory = method_data["b_theory"]
    b_impl = method_data["b_impl"]
    b_deploy = method_data["b_deploy"]

    # Validate bounds
    for name, value in [("b_theory", b_theory), ("b_impl", b_impl), ("b_deploy",
b_deploy)]:
        if not (0.0 <= value <= 1.0):
            raise ValueError(f"{name} must be in [0,1], got {value}")

    # Compute weighted sum
    score = (weights["w_th"] * b_theory +
             weights["w_imp"] * b_impl +
             weights["w_dep"] * b_deploy)

    return score


def compute_chain_layer(node_id: str, graph: ComputationGraph,
                        contextual_config: dict[str, Any]) -> float:
    """
    Compute chain compatibility layer (@chain)

    Spec compliance: Section 3.2
    Rule-based discrete mapping

    Args:
        node_id: Node identifier
        graph: Computation graph containing node
        contextual_config: Loaded contextual_parametrization.json

    Returns:
        Score in [0,1]
    """
    if node_id not in graph.nodes:
        raise ValueError(f"Node {node_id} not in graph")

    mappings = contextual_config["layer_chain"]["discrete_mappings"]

    # Check for hard mismatches (simplified - would need full schema validation)
```

```
    signature = graph.node_signatures.get(node_id, {})
    required_inputs = signature.get("required_inputs", [])

    # Simplified validation logic
    has_hard_mismatch = False

    # Check incoming edges for type compatibility
    incoming_edges = [e for e in graph.edges if e[1] == node_id]

    if not incoming_edges and required_inputs:
        has_hard_mismatch = True

    if has_hard_mismatch:
        return mappings["hard_mismatch"]
    # The following branches are unreachable because has_soft_violation and has_warnings
are never set to True.
    # If future logic is added to set these flags, restore these branches.
    else:
        return mappings["all_contracts_pass_no_warnings"]


def compute_unit_layer(method_id: str, role: MethodRole, unit_quality: float,
                       contextual_config: dict[str, Any]) -> float:
    """
    Compute unit-of-analysis sensitivity layer (@u)

    Spec compliance: Section 3.3
    Formula: x_@u = g_M(U) if M is U-sensitive, else 1.0

    Args:
        method_id: Canonical method ID
        role: Method role
        unit_quality: U in [0,1]
        contextual_config: Loaded contextual_parametrization.json

    Returns:
        Score in [0,1]
    """
    if not (0.0 <= unit_quality <= 1.0):
        raise ValueError(f"unit_quality must be in [0,1], got {unit_quality}")

    g_functions = contextual_config["layer_unit_of_analysis"]["g_functions"]
    role_name = role.value

    if role_name not in g_functions:
        # Default: not sensitive
        return 1.0

    g_spec = g_functions[role_name]
    g_type = g_spec["type"]

    if g_type == "identity":
        return unit_quality

    elif g_type == "constant":
        return 1.0

    elif g_type == "piecewise_linear":
        # g(U) = 2*U - 0.6 if U >= 0.3 else 0
        # Per canonic_calibration_methods.md: NO clamping - weights must be configured
correctly
        abort_threshold = g_spec.get("abort_threshold", 0.3)
        if unit_quality < abort_threshold:
            return 0.0
        score = 2.0 * unit_quality - 0.6

        # Validate that config produces valid result
        if score < 0.0 or score > 1.0:
```

```
            raise CalibrationConfigError(
                f"Unit layer g_function produced out-of-range score: {score} "
                f"for unit_quality={unit_quality}. Config must be adjusted to ensure [0,1]
 output."
            )
        return score

    elif g_type == "sigmoidal":
        # g(U) = 1 - exp(-k*(U - x0))
        # Per canonic_calibration_methods.md: NO clamping - config must produce [0,1]
        k = g_spec.get("sigmoidal_k", 5.0)
        x0 = g_spec.get("sigmoidal_x0", 0.5)
        score = 1.0 - math.exp(-k * (unit_quality - x0))

        # Validate that config produces valid result
        if score < 0.0 or score > 1.0:
            raise CalibrationConfigError(
                f"Unit layer g_function produced out-of-range score: {score} "
                f"for unit_quality={unit_quality}, k={k}, x0={x0}. "
                f"Config must be adjusted to ensure [0,1] output."
            )
        return score

    else:
        raise ValueError(f"Unknown g_function type: {g_type}")


def compute_question_layer(method_id: str, question_id: str | None,
                           monolith: dict[str, Any],
                           contextual_config: dict[str, Any]) -> float:
    """
    Compute question compatibility layer (@q)

    Spec compliance: Section 3.4
    Formula: x_@q = Q_f(M | Q)

    Args:
        method_id: Canonical method ID
        question_id: Question ID (or None)
        monolith: Loaded questionnaire_monolith.json
        contextual_config: Loaded contextual_parametrization.json

    Returns:
        Score in [0,1]
    """
    if question_id is None:
        # No specific question context
        return contextual_config["layer_question"]["compatibility_levels"]["undeclared"]

    levels = contextual_config["layer_question"]["compatibility_levels"]

    # Find question in monolith
    micro_questions = monolith.get("blocks", {}).get("micro_questions", [])
    question = None
    for q in micro_questions:
        if q.get("question_id") == question_id:
            question = q
            break

    if not question:
        return levels["undeclared"]

    # Check method_sets
    method_sets = question.get("method_sets", [])

    for method_spec in method_sets:
        # Match by function name or class name (simplified)
        if (method_id.endswith(f".{method_spec.get('function', '')}") or
```

```
            method_spec.get('class', '') in method_id):

            method_type = method_spec.get("method_type", "")
            priority = method_spec.get("priority", 99)

            if method_type == "extraction" or priority == 1:
                return levels["primary"]
            elif priority == 2:
                return levels["secondary"]
            elif method_type == "validation":
                return levels["validator"]

    return levels["undeclared"]


def compute_dimension_layer(method_id: str, dimension_id: str,
                            contextual_config: dict[str, Any]) -> float:
    """
    Compute dimension compatibility layer (@d)

    Spec compliance: Section 3.5
    Formula: x_@d = D_f(M | D)

    Args:
        method_id: Canonical method ID
        dimension_id: Dimension ID (DIM01-DIM06)
        contextual_config: Loaded contextual_parametrization.json

    Returns:
        Score in [0,1]
    """
    alignment = contextual_config["layer_dimension"]["alignment_matrix"]

    if dimension_id not in alignment:
        raise ValueError(f"Unknown dimension: {dimension_id}")

    dim_spec = alignment[dimension_id]

    # Simplified: return default score
    # Full implementation would check method family compatibility
    return dim_spec.get("default_score", 1.0)


def compute_policy_layer(method_id: str, policy_id: str,
                         contextual_config: dict[str, Any]) -> float:
    """
    Compute policy area compatibility layer (@p)

    Spec compliance: Section 3.6
    Formula: x_@p = P_f(M | P)

    Args:
        method_id: Canonical method ID
        policy_id: Policy area ID (PA01-PA10)
        contextual_config: Loaded contextual_parametrization.json

    Returns:
        Score in [0,1]
    """
    policies = contextual_config["layer_policy"]["policy_areas"]

    if policy_id not in policies:
        raise ValueError(f"Unknown policy area: {policy_id}")

    policy_spec = policies[policy_id]

    # Return default score (0.9 to satisfy anti-universality)
    return policy_spec.get("default_score", 0.9)
```

```
def compute_interplay_layer(interplay: InterplaySubgraph | None,
                            contextual_config: dict[str, Any]) -> float:
    """
    Compute interplay congruence layer (@C)

    Spec compliance: Section 3.7
    Formula: C_play(G | ctx) = c_scale · c_sem · c_fusion

    Args:
        interplay: Interplay subgraph (or None)
        contextual_config: Loaded contextual_parametrization.json

    Returns:
        Score in [0,1]
    """
    if interplay is None:
        # Not in an interplay
        return contextual_config["layer_interplay"]["default_when_not_in_interplay"]

    components = contextual_config["layer_interplay"]["components"]

    # Simplified computation
    c_scale = components["c_scale"]["same_range"]  # Assume same range
    c_sem = 1.0  # Assume full semantic overlap (simplified)
    c_fusion = components["c_fusion"]["declared_and_satisfied"]  # Assume declared

    return c_scale * c_sem * c_fusion


def compute_meta_layer(evidence: dict[str, Any],
                       contextual_config: dict[str, Any]) -> float:
    """
    Compute meta/governance layer (@m)

    Spec compliance: Section 3.8
    Formula: x_@m = 0.5 · m_transp + 0.4 · m_gov + 0.1 · m_cost

    Args:
        evidence: Evidence dictionary with metrics
        contextual_config: Loaded contextual_parametrization.json

    Returns:
        Score in [0,1]
    """
    meta_spec = contextual_config["layer_meta"]

    # Compute m_transp (transparency)
    transp_conditions = [
        evidence.get("formula_export_valid", False),
        evidence.get("trace_complete", False),
        evidence.get("logs_conform_schema", False)
    ]
    transp_count = sum(transp_conditions)

    transp_values = meta_spec["components"]["m_transp"]
    if transp_count == 3:
        m_transp = transp_values["all_three_conditions"]
    elif transp_count == 2:
        m_transp = transp_values["two_of_three"]
    elif transp_count == 1:
        m_transp = transp_values["one_of_three"]
    else:
        m_transp = transp_values["none"]

    # Compute m_gov (governance)
    gov_conditions = [
```

```
        evidence.get("version_tagged", False),
        evidence.get("config_hash_matches", False),
        evidence.get("signature_valid", False)
    ]
    gov_count = sum(gov_conditions)

    gov_values = meta_spec["components"]["m_gov"]
    if gov_count == 3:
        m_gov = gov_values["all_three_conditions"]
    elif gov_count == 2:
        m_gov = gov_values["two_of_three"]
    elif gov_count == 1:
        m_gov = gov_values["one_of_three"]
    else:
        m_gov = gov_values["none"]

    # Compute m_cost
    runtime_ms = evidence.get("runtime_ms", 100)
    thresholds = meta_spec["components"]["m_cost"]["thresholds"]

    if runtime_ms < thresholds["fast_runtime_ms"]:
        m_cost = meta_spec["components"]["m_cost"]["fast"]
    elif runtime_ms < thresholds["acceptable_runtime_ms"]:
        m_cost = meta_spec["components"]["m_cost"]["acceptable"]
    else:
        m_cost = meta_spec["components"]["m_cost"]["slow"]

    # Aggregate
    weights = meta_spec["aggregation"]["weights"]
    score = (weights["transparency"] * m_transp +
             weights["governance"] * m_gov +
             weights["cost"] * m_cost)

    return score

===== FILE: src/saaaaaa/core/calibration/layer_requirements.py =====
"""Layer Requirements Module.

This module defines the mandatory requirements for each method type (layer).
It is the SINGLE SOURCE OF TRUTH for deciding which layers a method must pass.
"""

from typing import List, Dict, Any
import logging
from .intrinsic_loader import IntrinsicCalibrationLoader

logger = logging.getLogger(__name__)

LAYER_REQUIREMENTS = {
    "ingest": {
        "layers": ["@b", "@chain", "@u", "@m"],
        "count": 4,
        "description": "Data ingestion - simple loading",
        "min_confidence": 0.5
    },

    "processor": {
        "layers": ["@b", "@chain", "@u", "@m"],
        "count": 4,
        "description": "Data processing - transformation without decisions",
        "min_confidence": 0.5
    },

    "analyzer": {
        "layers": ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"],
        "count": 8,
        "description": "Complex analysis - ALL context needed",
        "min_confidence": 0.7
```

```
    },

    "extractor": {
        "layers": ["@b", "@chain", "@u", "@m"],
        "count": 4,
        "description": "Feature extraction - pattern finding",
        "min_confidence": 0.5
    },

    "score": {
        "layers": ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"],
        "count": 8,
        "description": "Scoring methods (non-executor) - ALL context",
        "min_confidence": 0.7
    },

    "utility": {
        "layers": ["@b", "@chain", "@m"],
        "count": 3,
        "description": "Helpers - minimal validation",
        "min_confidence": 0.3
    },

    "orchestrator": {
        "layers": ["@b", "@chain", "@m"],
        "count": 3,
        "description": "Coordination - minimal validation",
        "min_confidence": 0.5
    },

    "core": {
        "layers": ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"],
        "count": 8,
        "description": "Critical foundation methods - ALL context",
        "min_confidence": 0.8
    }
}

# VALIDATION: Ensure consistency
assert all(
    len(config["layers"]) == config["count"]
    for config in LAYER_REQUIREMENTS.values()
), "Layer count mismatch in LAYER_REQUIREMENTS"


def get_required_layers_for_method(method_id: str) -> List[str]:
    """
    OBLIGATORY: Single function that decides layers for a method.

    NON-NEGOTIABLE:
    - SINGLE source of truth
    - NO overrides allowed
    - NO hardcoding elsewhere
    """

    # 1. Load intrinsic JSON via Singleton
    loader = IntrinsicCalibrationLoader()
    metadata = loader.get_metadata(method_id)

    # 2. If executor -> ALWAYS 8 layers (conservative default for executors)
    # Note: Ideally this should be driven by metadata, but keeping the rule as requested.
    if "executor" in method_id.lower():
        return ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"]

    # 3. Get "layer" from metadata
    if metadata is None:
        logger.warning(f"Method {method_id} not in intrinsic_calibration.json, using
conservative ALL layers")
```

```
        return ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"]

    method_type = metadata.get("layer")

    if method_type is None:
        logger.warning(f"Method {method_id} has no 'layer' field, using conservative ALL
layers")
        return ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"]

    # 4. Map to required layers
    if method_type not in LAYER_REQUIREMENTS:
        logger.error(f"Unknown method type '{method_type}' for {method_id}, using
conservative ALL layers")
        return ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"]

    required_layers = LAYER_REQUIREMENTS[method_type]["layers"]

    logger.debug(f"Method {method_id} (type: {method_type}) requires
{len(required_layers)} layers: {required_layers}")

    return required_layers

===== FILE: src/saaaaaa/core/calibration/meta_layer.py =====
"""
Meta Layer (@m) - Full Implementation.

Evaluates governance compliance using weighted formula:
x_@m = 0.5·m_transp + 0.4·m_gov + 0.1·m_cost
"""
import logging

from .config import MetaLayerConfig

logger = logging.getLogger(__name__)


class MetaLayerEvaluator:
    """
    Evaluates governance and meta-properties of methods.

    Attributes:
        config: MetaLayerConfig with weights and thresholds
    """

    def __init__(self, config: MetaLayerConfig) -> None:
        """
        Initialize evaluator with meta layer config.

        Args:
            config: MetaLayerConfig instance
        """
        self.config = config
        logger.info(
            "meta_evaluator_initialized",
            extra={
                "w_transparency": config.w_transparency,
                "w_governance": config.w_governance,
                "w_cost": config.w_cost
            }
        )

    def evaluate(
        self,
        method_id: str,
        method_version: str,
        config_hash: str,
        formula_exported: bool = False,
        full_trace: bool = False,
```

```
        logs_conform: bool = False,
        signature_valid: bool = False,
        execution_time_s: float | None = None
    ) -> float:
        """
        Compute the weighted score x_@m = w_transparency·m_transp + w_governance·m_gov +
w_cost·m_cost,
        where `w_transparency`, `w_governance`, and `w_cost` come from the provided
`config`.

        Args:
            method_id: Method identifier
            method_version: Method version string
            config_hash: Configuration hash
            formula_exported: Has formula been documented?
            full_trace: Is full execution trace available?
            logs_conform: Do logs conform to standard?
            signature_valid: Is cryptographic signature valid?
            execution_time_s: Runtime in seconds

        Returns:
            x_@m ∈ [0.0, 1.0]
        """
        logger.info(
            "meta_evaluation_start",
            extra={
                "method": method_id,
                "version": method_version
            }
        )

        # Component 1: Transparency (m_transp)
        m_transp = self._compute_transparency(
            formula_exported, full_trace, logs_conform
        )
        logger.debug("m_transp_computed", extra={"score": m_transp})

        # Component 2: Governance (m_gov)
        m_gov = self._compute_governance(
            method_version, config_hash, signature_valid
        )
        logger.debug("m_gov_computed", extra={"score": m_gov})

        # Component 3: Cost (m_cost)
        m_cost = self._compute_cost(execution_time_s)
        logger.debug("m_cost_computed", extra={"score": m_cost})

        # Weighted sum
        x_m = (
            self.config.w_transparency * m_transp +
            self.config.w_governance * m_gov +
            self.config.w_cost * m_cost
        )

        logger.info(
            "meta_computed",
            extra={
                "x_m": x_m,
                "m_transp": m_transp,
                "m_gov": m_gov,
                "m_cost": m_cost,
                "method": method_id
            }
        )

        return x_m

    def _compute_transparency(
```

```
    self,
    formula: bool,
    trace: bool,
    logs: bool
) -> float:
    """
    Compute m_transp based on observability.

    Scoring:
        1.0: All 3 conditions met
        0.7: 2/3 conditions met
        0.4: 1/3 conditions met
        0.0: 0/3 conditions met

    Returns:
        m_transp ∈ {0.0, 0.4, 0.7, 1.0}
    """
    count = sum([formula, trace, logs])

    if count == 3:
        return 1.0
    elif count == 2:
        return 0.7
    elif count == 1:
        return 0.4
    else:
        return 0.0

def _compute_governance(
    self,
    version: str,
    config_hash: str,
    signature: bool
) -> float:
    """
    Compute m_gov based on governance compliance.

    Scoring:
        1.0: All 3 conditions met
        0.66: 2/3 conditions met
        0.33: 1/3 conditions met
        0.0: 0/3 conditions met

    Returns:
        m_gov ∈ {0.0, 0.33, 0.66, 1.0}
    """
    # Check version
    has_version = bool(version and version not in {"unknown", "1.0"})

    # Check config hash
    has_hash = bool(config_hash and len(config_hash) > 0)

    # Count conditions
    count = sum([has_version, has_hash, signature])

    if count == 3:
        return 1.0
    elif count == 2:
        return 0.66
    elif count == 1:
        return 0.33
    else:
        return 0.0

def _compute_cost(self, execution_time_s: float | None = None) -> float:
    """
    Compute m_cost based on runtime.
```

```
        Scoring:
            1.0: < threshold_fast (e.g., <1s)
            0.8: < threshold_acceptable (e.g., <5s)
            0.5: >= threshold_acceptable
            0.0: timeout/OOM (not provided)

        Returns:
            m_cost ∈ {0.0, 0.5, 0.8, 1.0}
        """
        if execution_time_s is None:
            logger.warning("meta_timing_not_available")
            return 0.5  # Default: acceptable

        if execution_time_s < 0:
            logger.error("meta_negative_time", extra={"time": execution_time_s})
            return 0.0

        if execution_time_s < self.config.threshold_fast:
            return 1.0  # Fast
        elif execution_time_s < self.config.threshold_acceptable:
            return 0.8  # Acceptable
        else:
            logger.warning(
                "meta_slow_execution",
                extra={
                    "runtime": execution_time_s,
                    "threshold": self.config.threshold_acceptable
                }
            )
            return 0.5  # Slow but usable

===== FILE: src/saaaaaa/core/calibration/orchestrator.py =====
"""Calibration Orchestrator.

Central singleton that coordinates the calibration process.
"""

import logging
from typing import Any, Dict, List
from dataclasses import dataclass, field

from .intrinsic_loader import IntrinsicCalibrationLoader
from .layer_requirements import get_required_layers_for_method

logger = logging.getLogger(__name__)

@dataclass
class CalibrationResult:
    final_score: float
    layer_scores: Dict[str, float]
    metadata: Dict[str, Any]

    def get_failure_reason(self) -> str:
        if self.final_score < 0.5: # Simple threshold for reason
            return "Low confidence score"
        return ""

class CalibrationOrchestrator:
    _instance = None
    _initialized = False

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(CalibrationOrchestrator, cls).__new__(cls)
        return cls._instance

    def initialize(self) -> None:
        """Initialize the orchestrator and dependencies."""
```

```
        if self._initialized:
            return

        self.intrinsic_loader = IntrinsicCalibrationLoader()
        self.intrinsic_loader.load()
        self._initialized = True
        logger.info("CalibrationOrchestrator initialized")

    def calibrate(self, method_id: str, context: Dict[str, Any]) -> CalibrationResult:
        """
        Calibrate a method execution.

        1. Load @b from IntrinsicCalibrationLoader
        2. Determine layers from LAYER_REQUIREMENTS
        3. Evaluate each layer
        4. Aggregate
        5. Return CalibrationResult
        """
        if not self._initialized:
            self.initialize()

        # 1. Get Intrinsic Score (@b)
        intrinsic_score = self.intrinsic_loader.get_intrinsic_score(method_id)

        # 2. Determine Required Layers
        required_layers = get_required_layers_for_method(method_id)

        # 3. Evaluate Layers
        # In a full implementation, this would delegate to specific Layer classes
(BaseLayerEvaluator, etc.)
        # For now, we ensure @b is correct and others are estimated relative to it or
context
        layer_scores = {}
        for layer in required_layers:
            if layer == "@b":
                layer_scores[layer] = intrinsic_score
            else:
                # Placeholder: In the future, instantiate specific layer evaluators here.
                # For now, we assume if @b is high, others are likely high, but we cap
them.
                # This allows the system to function without full implementation of all 8
layers.
                layer_scores[layer] = min(1.0, intrinsic_score * 1.1)

        # 4. Aggregate (Simple average for now, Choquet in full implementation)
        if layer_scores:
            final_score = sum(layer_scores.values()) / len(layer_scores)
        else:
            final_score = intrinsic_score

        return CalibrationResult(
            final_score=final_score,
            layer_scores=layer_scores,
            metadata={"method_id": method_id, "layers_evaluated": required_layers}
        )

===== FILE: src/saaaaaa/core/calibration/parameter_loader.py =====
"""Parameter Loader.

Singleton loader for method_parameters.json (System 1).
"""

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

logger = logging.getLogger(__name__)
```

```
class ParameterLoader:
    _instance = None
    _data: Dict[str, Any] = {}
    _loaded = False

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(ParameterLoader, cls).__new__(cls)
        return cls._instance

    def load(self, config_path: str = "config/method_parameters.json") -> None:
        """Load method parameters from JSON."""
        if self._loaded:
            return

        path = Path(config_path)
        if not path.exists():
            # Try finding it relative to repo root
            repo_root = Path(__file__).parent.parent.parent.parent.parent
            path = repo_root / config_path

        if not path.exists():
             logger.warning(f"Method parameters file not found at {path}. Using empty
config.")
             self._data = {}
             self._loaded = True
             return

        try:
            with open(path, "r") as f:
                self._data = json.load(f)
            self._loaded = True
            logger.info(f"Loaded method parameters from {path}")
        except Exception as e:
            logger.error(f"Failed to load method parameters: {e}")
            self._data = {}

    def get(self, method_id: str) -> Dict[str, Any]:
        """Get parameters for a method."""
        if not self._loaded:
            self.load()
        return self._data.get(method_id, {})

===== FILE: src/saaaaaa/core/calibration/pdt_structure.py =====
"""
PDT (Plan de Desarrollo Territorial) structure definition.

This module defines the data structure that represents a parsed PDT.
The PDT is the INPUT to the Unit Layer evaluation.

The structure is populated by a separate PDT parser (not shown here),
which extracts:
- Text content and tokens
- Block structure (Diagnóstico, Estratégica, PPI, Seguimiento)
- Section analysis (keywords, numbers, sources)
- Indicator matrix (if present)
- PPI matrix (if present)
"""
from dataclasses import dataclass, field
from typing import Any


@dataclass
class PDTStructure:
    """
    Extracted structure of a PDT.
```

```
This is populated by the PDT parser and consumed by UnitLayerEvaluator.

Attributes:
    full_text: Complete text of the PDT
    total_tokens: Total word/token count
    blocks_found: Detected structural blocks
    headers: List of headers with numbering validation
    block_sequence: Actual order of blocks (for checking sequence)
    sections_found: Analysis of mandatory sections
    indicator_matrix_present: Whether indicator table was found
    indicator_rows: Parsed indicator table rows
    ppi_matrix_present: Whether PPI table was found
    ppi_rows: Parsed PPI table rows
"""
# Raw content
full_text: str
total_tokens: int

# Block detection (for S - Structural compliance)
blocks_found: dict[str, dict[str, Any]] = field(default_factory=dict)
# Example: {
#     "Diagnóstico": {"text": "...", "tokens": 1500, "numbers_count": 25},
#     "Parte Estratégica": {"text": "...", "tokens": 1200, "numbers_count": 15},
# }

headers: list[dict[str, Any]] = field(default_factory=list)
# Example: [
#     {"level": 1, "text": "1. DIAGNÓSTICO", "valid_numbering": True},
#     {"level": 2, "text": "1.1 Contexto", "valid_numbering": True},
# ]

block_sequence: list[str] = field(default_factory=list)
# Example: ["Diagnóstico", "Parte Estratégica", "PPI", "Seguimiento"]

# Section analysis (for M - Mandatory sections)
sections_found: dict[str, dict[str, Any]] = field(default_factory=dict)
# Example: {
#     "Diagnóstico": {
#         "present": True,
#         "token_count": 1500,
#         "keyword_matches": 5,  # e.g., "brecha", "DANE", "línea base"
#         "number_count": 25,
#         "sources_found": 3,  # e.g., "DANE", "Medicina Legal"
#     }
# }

# Indicator matrix (for I - Indicator quality)
indicator_matrix_present: bool = False
indicator_rows: list[dict[str, Any]] = field(default_factory=list)
# Example: [
#     {
#         "Tipo": "PRODUCTO",
#         "Línea Estratégica": "Equidad de Género",
#         "Programa": "Prevención de VBG",
#         "Línea Base": "120 casos",
#         "Año LB": 2023,
#         "Meta Cuatrienio": "80 casos",
#         "Fuente": "Comisaría de Familia",
#         "Unidad Medida": "Casos reportados",
#         "Código MGA": "1234567"
#     }
# ]

# PPI matrix (for P - PPI completeness)
ppi_matrix_present: bool = False
ppi_rows: list[dict[str, Any]] = field(default_factory=list)
# Example: [
#     {
```

```
    #         "Línea Estratégica": "Equidad de Género",
    #         "Programa": "Prevención de VBG",
    #         "Costo Total": 500000000,
    #         "2024": 100000000,
    #         "2025": 150000000,
    #         "2026": 150000000,
    #         "2027": 100000000,
    #         "SGP": 300000000,
    #         "SGR": 0,
    #         "Propios": 200000000,
    #         "Otras": 0
    #     }
    # ]

===== FILE: src/saaaaaa/core/calibration/protocols.py =====
"""
Calibration system protocols and abstract interfaces.

This module defines the contracts that all layer evaluators must satisfy.
Using Protocol instead of ABC allows for structural subtyping (duck typing)
while still enabling static type checking with mypy.

Design Philosophy:
- All evaluators return LayerScore (not raw float)
- Evaluation methods must be pure (no side effects beyond logging)
- All parameters must be explicitly typed
- Protocols enable static verification without inheritance
"""
from typing import TYPE_CHECKING, Protocol, runtime_checkable

from .data_structures import LayerScore

if TYPE_CHECKING:
    from .pdt_structure import PDTStructure


@runtime_checkable
class LayerEvaluator(Protocol):
    """
    Protocol that all layer evaluators must implement.

    This ensures consistent signature across all evaluation layers:
    - BASE (@b)
    - UNIT (@u)
    - QUESTION (@q)
    - DIMENSION (@d)
    - POLICY (@p)
    - CONGRUENCE (@C)
    - CHAIN (@chain)
    - META (@m)

    By using Protocol, we enable static type checking without requiring
    explicit inheritance. Any class with an `evaluate()` method that
    returns LayerScore will satisfy this protocol.

    Example:
        def process_layer(evaluator: LayerEvaluator, **kwargs):
            score = evaluator.evaluate(**kwargs)
            assert isinstance(score, LayerScore)  # Always true
    """

    def evaluate(self, **kwargs) -> LayerScore:
        """
        Evaluate layer and return LayerScore.

        Each evaluator can accept different keyword arguments depending
        on what it needs, but MUST return LayerScore.
```

```
        Returns:
            LayerScore with score ∈ [0.0, 1.0], components, and rationale

        Example implementations:
            # BASE layer
            def evaluate(self, method_id: str) -> LayerScore: ...

            # UNIT layer
            def evaluate(self, pdt: PDTStructure) -> LayerScore: ...

            # QUESTION layer
            def evaluate(self, method_id: str, question_id: str) -> LayerScore: ...
        """
        ...


@runtime_checkable
class BaseLayerEvaluatorProtocol(Protocol):
    """
    Specific protocol for BASE layer evaluation.

    BASE layer evaluates intrinsic method quality from pre-computed
    calibration scores (b_theory, b_impl, b_deploy).
    """

    def evaluate(self, method_id: str) -> LayerScore:
        """
        Evaluate BASE layer for a method.

        Args:
            method_id: Canonical method identifier

        Returns:
            LayerScore with @b score and component breakdown
        """
        ...


@runtime_checkable
class UnitLayerEvaluatorProtocol(Protocol):
    """
    Specific protocol for UNIT layer evaluation.

    UNIT layer evaluates PDT quality through 4 components: S, M, I, P.
    """

    def evaluate(self, pdt: "PDTStructure") -> LayerScore:  # type: ignore
        """
        Evaluate UNIT layer from PDT structure.

        Args:
            pdt: Parsed PDT structure

        Returns:
            LayerScore with U score and SMIP components
        """
        ...


@runtime_checkable
class ContextualLayerEvaluatorProtocol(Protocol):
    """
    Protocol for contextual layers (@q, @d, @p).

    These layers evaluate method-context compatibility.
    """

    def evaluate_question(self, method_id: str, question_id: str) -> float:
```

```
        """Evaluate @q (question compatibility)."""
        ...

    def evaluate_dimension(self, method_id: str, dimension: str) -> float:
        """Evaluate @d (dimension compatibility)."""
        ...

    def evaluate_policy(self, method_id: str, policy_area: str) -> float:
        """Evaluate @p (policy compatibility)."""
        ...


@runtime_checkable
class CongruenceLayerEvaluatorProtocol(Protocol):
    """
    Protocol for CONGRUENCE layer (@C).

    Evaluates ensemble validity when multiple methods work together.
    """

    def evaluate(
        self,
        method_ids: list[str],
        subgraph_id: str,
        fusion_rule: str,
        available_inputs: list[str]
    ) -> float:
        """
        Evaluate congruence for a method ensemble.

        Args:
            method_ids: List of methods in the ensemble
            subgraph_id: Identifier for the interaction subgraph
            fusion_rule: How outputs are combined (e.g., "weighted_average")
            available_inputs: Inputs available to the ensemble

        Returns:
            Congruence score ∈ [0.0, 1.0]
        """
        ...


@runtime_checkable
class ChainLayerEvaluatorProtocol(Protocol):
    """
    Protocol for CHAIN layer (@chain).

    Evaluates data flow integrity between method inputs and outputs.
    """

    def evaluate(
        self,
        method_id: str,
        provided_inputs: list[str]
    ) -> float:
        """
        Evaluate chain integrity for a method.

        Args:
            method_id: Method identifier
            provided_inputs: Inputs provided to the method

        Returns:
            Chain score ∈ [0.0, 1.0]
        """
        ...
```

```
@runtime_checkable
class MetaLayerEvaluatorProtocol(Protocol):
    """
    Protocol for META layer (@m).

    Evaluates governance, transparency, and computational cost.
    """

    def evaluate(
        self,
        method_id: str,
        method_version: str,
        config_hash: str,
        formula_exported: bool,
        full_trace: bool,
        logs_conform: bool,
        signature_valid: bool,
        execution_time_s: float | None
    ) -> float:
        """
        Evaluate meta/governance layer.

        Args:
            method_id: Method identifier
            method_version: Semantic version
            config_hash: Configuration hash for reproducibility
            formula_exported: Whether computation formula is exported
            full_trace: Whether full execution trace is available
            logs_conform: Whether logs conform to schema
            signature_valid: Whether cryptographic signature is valid
            execution_time_s: Execution time in seconds (None if not measured)

        Returns:
            Meta score ∈ [0.0, 1.0]
        """
        ...


def validate_evaluator_protocol(evaluator: object, protocol_type: type) -> bool:
    """
    Validate that an evaluator satisfies a protocol.

    This is useful for runtime validation before using an evaluator.

    Args:
        evaluator: The evaluator instance to check
        protocol_type: The protocol type to check against (e.g., LayerEvaluator)

    Returns:
        True if evaluator satisfies protocol, False otherwise

    Example:
        evaluator = BaseLayerEvaluator("config/intrinsic_calibration.json")
        assert validate_evaluator_protocol(evaluator, LayerEvaluator)
    """
    return isinstance(evaluator, protocol_type)

===== FILE: src/saaaaaa/core/calibration/unit_layer.py =====
"""
Unit Layer (@u) - PRODUCTION IMPLEMENTATION.

Evaluates PDT quality through 4 components: S, M, I, P.
"""
import logging

from .config import UnitLayerConfig
from .data_structures import LayerID, LayerScore
from .pdt_structure import PDTStructure
```

```
logger = logging.getLogger(__name__)


class UnitLayerEvaluator:
    """
    Evaluates Unit Layer (@u) - PDT quality.

    PRODUCTION IMPLEMENTATION - All scores are data-driven.
    """

    # Mandatory blocks required for PDT compliance
    MANDATORY_BLOCKS = ["Diagnóstico", "Parte Estratégica", "PPI", "Seguimiento"]

    def __init__(self, config: UnitLayerConfig) -> None:
        self.config = config

    def evaluate(self, pdt: PDTStructure) -> LayerScore:
        """
        Production implementation - computes S, M, I, P from PDT data.

        THIS IS NOT A STUB - all scores are data-driven.
        """
        logger.info("unit_layer_evaluation_start", extra={"tokens": pdt.total_tokens})

        # Step 1: Compute S (Structural Compliance)
        S = self._compute_structural_compliance(pdt)
        logger.info("S_computed", extra={"S": S})

        # Step 2: Check hard gate for S
        if self.config.min_structural_compliance > S:
            return LayerScore(
                layer=LayerID.UNIT,
                score=0.0,
                components={"S": S, "gate_failure": "structural"},
                rationale=f"HARD GATE: S={S:.2f} <
{self.config.min_structural_compliance}",
                metadata={"gate": "structural", "threshold":
self.config.min_structural_compliance}
            )

        # Step 3: Compute M (Mandatory Sections)
        M = self._compute_mandatory_sections(pdt)
        logger.info("M_computed", extra={"M": M})

        # Step 4: Compute I (Indicator Quality)
        I_components = self._compute_indicator_quality(pdt)
        I = I_components["I_total"]
        logger.info("I_computed", extra={"I": I})

        # Step 5: Check hard gate for I_struct
        if I_components["I_struct"] < self.config.i_struct_hard_gate:
            return LayerScore(
                layer=LayerID.UNIT,
                score=0.0,
                components={"S": S, "M": M, "I_struct": I_components["I_struct"]},
                rationale=f"HARD GATE: I_struct={I_components['I_struct']:.2f} <
{self.config.i_struct_hard_gate}",
                metadata={"gate": "indicator_structure"}
            )

        # Step 6: Compute P (PPI Completeness)
        P_components = self._compute_ppi_completeness(pdt)
        P = P_components["P_total"]
        logger.info("P_computed", extra={"P": P})

        # Step 7: Check hard gates for PPI
        if self.config.require_ppi_presence and not pdt.ppi_matrix_present:
```

```
            return LayerScore(
                layer=LayerID.UNIT,
                score=0.0,
                components={"S": S, "M": M, "I": I, "gate_failure": "ppi_presence"},
                rationale="HARD GATE: PPI required but not present",
                metadata={"gate": "ppi_presence"}
            )

        if self.config.require_indicator_matrix and not pdt.indicator_matrix_present:
            return LayerScore(
                layer=LayerID.UNIT,
                score=0.0,
                components={"S": S, "M": M, "I": I, "P": P, "gate_failure":
"indicator_matrix"},
                rationale="HARD GATE: Indicator matrix required but not present",
                metadata={"gate": "indicator_matrix"}
            )

        # Step 8: Aggregate
        U_base = self._aggregate_components(S, M, I, P)
        logger.info("U_base_computed", extra={"U_base": U_base})

        # Step 9: Anti-gaming
        gaming_penalty = self._compute_gaming_penalty(pdt)
        U_final = max(0.0, U_base - gaming_penalty)

        # Step 10: Quality level
        if U_final >= 0.85:
            quality = "sobresaliente"
        elif U_final >= 0.7:
            quality = "robusto"
        elif U_final >= 0.5:
            quality = "mínimo"
        else:
            quality = "insuficiente"

        return LayerScore(
            layer=LayerID.UNIT,
            score=U_final,
            components={"S": S, "M": M, "I": I, "P": P, "U_base": U_base, "penalty":
gaming_penalty},
            rationale=f"Unit quality: {quality} (S={S:.2f}, M={M:.2f}, I={I:.2f},
P={P:.2f})",
            metadata={"quality_level": quality, "aggregation":
self.config.aggregation_type}
        )

    def _compute_structural_compliance(self, pdt: PDTStructure) -> float:
        """Compute S = w_block·B_cov + w_hierarchy·H + w_order·O."""
        # Block coverage
        blocks_found = sum(
            1 for block in self.MANDATORY_BLOCKS
            if block in pdt.blocks_found
            and pdt.blocks_found[block].get("tokens", 0) >= self.config.min_block_tokens
            and pdt.blocks_found[block].get("numbers_count", 0) >=
self.config.min_block_numbers
        )
        B_cov = blocks_found / len(self.MANDATORY_BLOCKS)

        # Hierarchy score
        if not pdt.headers:
            H = 0.0
        else:
            valid = sum(1 for h in pdt.headers if h.get("valid_numbering"))
            ratio = valid / len(pdt.headers)
            if ratio >= self.config.hierarchy_excellent_threshold:
                H = 1.0
            elif ratio >= self.config.hierarchy_acceptable_threshold:
```

```
            H = 0.5
        else:
            H = 0.0

    # Order score - count inversions in block_sequence vs expected
    expected = ["Diagnóstico", "Parte Estratégica", "PPI", "Seguimiento"]
    inversions = 0
    if pdt.block_sequence:
        # Find positions of blocks in actual sequence
        positions = {}
        for i, block in enumerate(pdt.block_sequence):
            if block in expected:
                positions[block] = i

        # Count inversions (pairs out of order)
        for i, block1 in enumerate(expected):
            if block1 not in positions:
                continue
            for block2 in expected[i+1:]:
                if block2 not in positions:
                    continue
                if positions[block1] > positions[block2]:
                    inversions += 1

    O = 1.0 if inversions == 0 else (0.5 if inversions == 1 else 0.0)

    S = (self.config.w_block_coverage * B_cov +
         self.config.w_hierarchy * H +
         self.config.w_order * O)

    return S

def _compute_mandatory_sections(self, pdt: PDTStructure) -> float:
    """Compute M = weighted average of section completeness."""
    # Section requirements (from config)
    requirements = {
        "Diagnóstico": {
            "min_tokens": self.config.diagnostico_min_tokens,
            "min_keywords": self.config.diagnostico_min_keywords,
            "min_numbers": self.config.diagnostico_min_numbers,
            "min_sources": self.config.diagnostico_min_sources,
            "weight": self.config.critical_sections_weight,  # Critical section
        },
        "Parte Estratégica": {
            "min_tokens": self.config.estrategica_min_tokens,
            "min_keywords": self.config.estrategica_min_keywords,
            "min_numbers": self.config.estrategica_min_numbers,
            "weight": self.config.critical_sections_weight,  # Critical section
        },
        "PPI": {
            "min_tokens": self.config.ppi_section_min_tokens,
            "min_keywords": self.config.ppi_section_min_keywords,
            "min_numbers": self.config.ppi_section_min_numbers,
            "weight": self.config.critical_sections_weight,  # Critical section
        },
        "Seguimiento": {
            "min_tokens": self.config.seguimiento_min_tokens,
            "min_keywords": self.config.seguimiento_min_keywords,
            "min_numbers": self.config.seguimiento_min_numbers,
            "weight": 1.0,
        },
        "Marco Normativo": {
            "min_tokens": self.config.marco_normativo_min_tokens,
            "min_keywords": self.config.marco_normativo_min_keywords,
            "weight": 1.0,
        }
    }
```

```
    total_weight = 0.0
    weighted_score = 0.0

    for section_name, reqs in requirements.items():
        section_data = pdt.sections_found.get(section_name, {})

        if not section_data.get("present", False):
            # Missing section gets 0
            score = 0.0
        else:
            # Check all requirements
            checks_passed = 0
            checks_total = 0

            if "min_tokens" in reqs:
                checks_total += 1
                if section_data.get("token_count", 0) >= reqs["min_tokens"]:
                    checks_passed += 1

            if "min_keywords" in reqs:
                checks_total += 1
                if section_data.get("keyword_matches", 0) >= reqs["min_keywords"]:
                    checks_passed += 1

            if "min_numbers" in reqs:
                checks_total += 1
                if section_data.get("number_count", 0) >= reqs["min_numbers"]:
                    checks_passed += 1

            if "min_sources" in reqs:
                checks_total += 1
                if section_data.get("sources_found", 0) >= reqs["min_sources"]:
                    checks_passed += 1

            score = checks_passed / checks_total if checks_total > 0 else 0.0

        weight = reqs.get("weight", 1.0)
        weighted_score += score * weight
        total_weight += weight

    M = weighted_score / total_weight if total_weight > 0 else 0.0
    return M

def _compute_indicator_quality(self, pdt: PDTStructure) -> dict:
    """Compute I = w_struct·I_struct + w_link·I_link + w_logic·I_logic."""
    if not pdt.indicator_matrix_present or not pdt.indicator_rows:
        logger.warning("indicator_matrix_absent", extra={"I": 0.0})
        return {
            "I_struct": 0.0,
            "I_link": 0.0,
            "I_logic": 0.0,
            "I_total": 0.0
        }

    # I_struct: Field completeness
    critical_fields = ["Tipo", "Línea Estratégica", "Programa", "Línea Base",
                       "Meta Cuatrienio", "Fuente", "Unidad Medida"]
    optional_fields = ["Año LB", "Código MGA"]

    total_struct_score = 0.0
    for row in pdt.indicator_rows:
        critical_present = sum(1 for f in critical_fields if row.get(f))
        optional_present = sum(1 for f in optional_fields if row.get(f))

        # Penalize placeholders
        placeholder_count = sum(
            1 for f in critical_fields
            if row.get(f) in ["S/D", "N/A", "TBD", ""]
```

```
            )

            critical_score = critical_present / len(critical_fields)
            optional_score = optional_present / len(optional_fields)
            placeholder_penalty = (placeholder_count / len(critical_fields)) *
self.config.i_placeholder_penalty_multiplier

            row_score = (critical_score * self.config.i_critical_fields_weight +
optional_score) / (self.config.i_critical_fields_weight + 1)
            row_score = max(0.0, row_score - placeholder_penalty)
            total_struct_score += row_score

        I_struct = total_struct_score / len(pdt.indicator_rows)

        # I_link: Traceability (fuzzy matching between indicators and strategic lines)
        linked_count = 0
        for row in pdt.indicator_rows:
            programa = row.get("Programa", "")
            linea = row.get("Línea Estratégica", "")
            if programa and linea:
                # Simplified: check if they share significant words
                prog_words = set(programa.lower().split())
                linea_words = set(linea.lower().split())
                if len(prog_words & linea_words) >= 2:  # At least 2 words in common
                    linked_count += 1

        I_link = linked_count / len(pdt.indicator_rows)

        # I_logic: Year coherence
        logic_violations = 0
        for row in pdt.indicator_rows:
            year_lb = row.get("Año LB")

            if year_lb is not None:
                try:
                    year_lb_int = int(year_lb)
                    if not (self.config.i_valid_lb_year_min <= year_lb_int <=
self.config.i_valid_lb_year_max):
                        logic_violations += 1
                except (ValueError, TypeError):
                    # Invalid year format counts as violation
                    logic_violations += 1

        I_logic = 1.0 - (logic_violations / len(pdt.indicator_rows))

        # Aggregate
        I_total = (self.config.w_i_struct * I_struct +
                   self.config.w_i_link * I_link +
                   self.config.w_i_logic * I_logic)

        return {
            "I_struct": I_struct,
            "I_link": I_link,
            "I_logic": I_logic,
            "I_total": I_total
        }

    def _compute_ppi_completeness(self, pdt: PDTStructure) -> dict:
        """Compute P = w_presence·P_presence + w_struct·P_struct +
w_cons·P_consistency."""
        # P_presence
        P_presence = 1.0 if pdt.ppi_matrix_present else 0.0

        if not pdt.ppi_matrix_present or not pdt.ppi_rows:
            return {
                "P_presence": P_presence,
                "P_struct": 0.0,
                "P_consistency": 0.0,
```

```
            "P_total": P_presence * self.config.w_p_presence
        }

    # P_struct: Non-zero rows
    nonzero_rows = sum(
        1 for row in pdt.ppi_rows
        if row.get("Costo Total", 0) > 0
    )
    P_struct = nonzero_rows / len(pdt.ppi_rows)

    # P_consistency: Accounting closure
    violations = 0
    for row in pdt.ppi_rows:
        costo_total = row.get("Costo Total", 0)

        # Check temporal sum
        temporal_sum = sum(row.get(str(year), 0) for year in range(2024, 2028))
        if abs(temporal_sum - costo_total) > costo_total *
self.config.p_accounting_tolerance:
            violations += 1

        # Check source sum
        source_sum = (row.get("SGP", 0) + row.get("SGR", 0) +
                row.get("Propios", 0) + row.get("Otras", 0))
        if abs(source_sum - costo_total) > costo_total *
self.config.p_accounting_tolerance:
            violations += 1

    P_consistency = 1.0 - (violations / (len(pdt.ppi_rows) * 2))  # 2 checks per row

    # Aggregate
    P_total = (self.config.w_p_presence * P_presence +
            self.config.w_p_structure * P_struct +
            self.config.w_p_consistency * P_consistency)

    return {
        "P_presence": P_presence,
        "P_struct": P_struct,
        "P_consistency": P_consistency,
        "P_total": P_total
    }

def _aggregate_components(self, S: float, M: float, I: float, P: float) -> float:
    """Aggregate S, M, I, P using configured method."""
    if self.config.aggregation_type == "geometric_mean":
        # Geometric mean: (S·M·I·P)^(1/4)
        product = S * M * I * P
        return product ** 0.25
    elif self.config.aggregation_type == "harmonic_mean":
        # Harmonic mean: 4 / (1/S + 1/M + 1/I + 1/P)
        if S == 0 or M == 0 or I == 0 or P == 0:
            return 0.0
        return 4.0 / (1.0/S + 1.0/M + 1.0/I + 1.0/P)
    else:  # weighted_average
        return (self.config.w_S * S +
                self.config.w_M * M +
                self.config.w_I * I +
                self.config.w_P * P)

def _compute_gaming_penalty(self, pdt: PDTStructure) -> float:
    """Compute anti-gaming penalties."""
    penalties = []

    # Check placeholder ratio in indicators
    if pdt.indicator_matrix_present and pdt.indicator_rows:
        placeholder_count = 0
        total_fields = 0
        for row in pdt.indicator_rows:
```

```
            for _key, value in row.items():
                total_fields += 1
                if value in ["S/D", "N/A", "TBD", ""]:
                    placeholder_count += 1

        placeholder_ratio = placeholder_count / total_fields if total_fields > 0 else
0
        if placeholder_ratio > self.config.max_placeholder_ratio:
            penalty = (placeholder_ratio - self.config.max_placeholder_ratio) * 0.5
            penalties.append(penalty)

    # Check unique values in PPI costs
    if pdt.ppi_matrix_present and pdt.ppi_rows:
        costs = [row.get("Costo Total", 0) for row in pdt.ppi_rows]
        unique_costs = len(set(costs))
        unique_ratio = unique_costs / len(costs) if costs else 0

        if unique_ratio < self.config.min_unique_values_ratio:
            penalty = (self.config.min_unique_values_ratio - unique_ratio) * 0.3
            penalties.append(penalty)

    # Check number density in critical sections
    critical_sections = ["Diagnóstico", "Parte Estratégica", "PPI"]
    for section in critical_sections:
        section_data = pdt.sections_found.get(section, {})
        if section_data.get("present"):
            tokens = section_data.get("token_count", 0)
            numbers = section_data.get("number_count", 0)
            density = numbers / tokens if tokens > 0 else 0

            if density < self.config.min_number_density:
                penalty = (self.config.min_number_density - density) * 0.2
                penalties.append(penalty)

    total_penalty = sum(penalties)
    return min(total_penalty, self.config.gaming_penalty_cap)

===== FILE: src/saaaaaa/core/calibration/validator.py =====
"""
Method validation system using calibration scores.

This module implements automatic validation that uses calibration scores
to make PASS/FAIL decisions about method execution.

Design:
- Integrates CalibrationOrchestrator for score computation
- Loads thresholds from MethodParameterLoader
- Makes validation decisions based on score vs threshold
- Provides detailed failure analysis
- Generates comprehensive validation reports
"""
import logging
from datetime import datetime
from typing import Dict, Optional, List, Any
from dataclasses import dataclass, field
from enum import Enum

from .orchestrator import CalibrationOrchestrator
from .parameter_loader import ParameterLoader
from .data_structures import CalibrationResult, ContextTuple, LayerID
from .pdt_structure import PDTStructure
from .intrinsic_loader import IntrinsicCalibrationLoader

logger = logging.getLogger(__name__)


class ValidationDecision(str, Enum):
    """Validation decision outcomes."""
```

```
    PASS = "PASS"
    FAIL = "FAIL"
    CONDITIONAL_PASS = "CONDITIONAL_PASS"
    SKIPPED = "SKIPPED"


class FailureReason(str, Enum):
    """Reasons for validation failure."""
    SCORE_BELOW_THRESHOLD = "score_below_threshold"
    BASE_LAYER_LOW = "base_layer_low_quality"
    CHAIN_LAYER_FAIL = "chain_layer_missing_inputs"
    CONGRUENCE_FAIL = "congruence_layer_inconsistent"
    UNIT_LAYER_FAIL = "unit_layer_pdt_quality_low"
    CONTEXTUAL_FAIL = "contextual_layer_incompatible"
    META_LAYER_FAIL = "meta_layer_governance_fail"
    METHOD_EXCLUDED = "method_excluded_from_calibration"
    UNKNOWN = "unknown"


@dataclass
class ValidationResult:
    """Result of a single method validation."""
    method_id: str
    decision: ValidationDecision
    calibration_score: float
    threshold: float
    timestamp: str
    calibration_result: Optional[CalibrationResult] = None
    failure_reason: Optional[FailureReason] = None
    failure_details: Optional[str] = None
    layer_scores: Dict[str, float] = field(default_factory=dict)
    recommendations: List[str] = field(default_factory=list)

    def to_dict(self) -> Dict[str, Any]:
        """Convert to dictionary for serialization."""
        return {
            "method_id": self.method_id,
            "decision": self.decision.value,
            "calibration_score": self.calibration_score,
            "threshold": self.threshold,
            "timestamp": self.timestamp,
            "failure_reason": self.failure_reason.value if self.failure_reason else None,
            "failure_details": self.failure_details,
            "layer_scores": self.layer_scores,
            "recommendations": self.recommendations
        }


@dataclass
class ValidationReport:
    """Comprehensive validation report for multiple methods."""
    plan_id: str
    timestamp: str
    total_methods: int
    passed: int
    failed: int
    conditional_pass: int
    skipped: int
    method_results: List[ValidationResult] = field(default_factory=list)
    overall_decision: ValidationDecision = ValidationDecision.CONDITIONAL_PASS
    summary: str = ""

    def pass_rate(self) -> float:
        """Calculate pass rate (0.0 - 1.0)."""
        if self.total_methods == 0:
            return 0.0
        return self.passed / self.total_methods
```

```
    def to_dict(self) -> Dict[str, Any]:
        """Convert to dictionary for serialization."""
        return {
            "plan_id": self.plan_id,
            "timestamp": self.timestamp,
            "total_methods": self.total_methods,
            "passed": self.passed,
            "failed": self.failed,
            "conditional_pass": self.conditional_pass,
            "skipped": self.skipped,
            "pass_rate": self.pass_rate(),
            "overall_decision": self.overall_decision.value,
            "summary": self.summary,
            "method_results": [r.to_dict() for r in self.method_results]
        }


class CalibrationValidator:
    """
    Validates methods using calibration scores.

    This is the main entry point for automatic validation based on calibration.

    Usage:
        validator = CalibrationValidator(
            orchestrator=calibration_orchestrator,
            parameter_loader=parameter_loader
        )

        # Validate a single method
        result = validator.validate_method(
            method_id="D1Q1_Executor",
            method_version="1.0.0",
            context=context,
            pdt_structure=pdt
        )

        if result.decision == ValidationDecision.FAIL:
            print(f"Method failed: {result.failure_reason}")

        # Validate all executors for a plan
        report = validator.validate_plan_executors(
            plan_id="plan_bogota_2024",
            context=context,
            pdt_structure=pdt
        )

        print(f"Plan validation: {report.overall_decision}")
        print(f"Pass rate: {report.pass_rate():.1%}")
    """

    def __init__(
        self,
        orchestrator: CalibrationOrchestrator,
        parameter_loader: ParameterLoader,
        intrinsic_loader: Optional[IntrinsicCalibrationLoader] = None
    ):
        """
        Initialize the validator.

        Args:
            orchestrator: Calibration orchestrator for computing scores
            parameter_loader: Loader for method parameters and thresholds
            intrinsic_loader: Optional intrinsic score loader (for exclusion checks)
        """
        self.orchestrator = orchestrator
        self.parameter_loader = parameter_loader
        self.intrinsic_loader = intrinsic_loader or orchestrator.intrinsic_loader
```

```
        logger.info("calibration_validator_initialized")

    def validate_method(
        self,
        method_id: str,
        method_version: str,
        context: ContextTuple,
        pdt_structure: PDTStructure,
        graph_config: str = "default",
        subgraph_id: str = "default",
        override_threshold: Optional[float] = None
    ) -> ValidationResult:
        """
        Validate a single method using calibration.

        Args:
            method_id: Method identifier
            method_version: Method version
            context: Execution context
            pdt_structure: PDT structure for unit layer
            graph_config: Computational graph hash
            subgraph_id: Subgraph identifier
            override_threshold: Optional threshold override (for testing)

        Returns:
            ValidationResult with decision and details
        """
        timestamp = datetime.utcnow().isoformat()

        canonical_method_id = self._resolve_canonical_method_id(method_id)

        # Check if method is excluded from calibration
        if self.intrinsic_loader and
self.intrinsic_loader.is_excluded(canonical_method_id):
            logger.info(
                "method_excluded_from_calibration",
                extra={
                    "method_id": method_id,
                    "canonical_method_id": canonical_method_id,
                }
            )
            return ValidationResult(
                method_id=method_id,
                decision=ValidationDecision.SKIPPED,
                calibration_score=1.0,  # Neutral, doesn't penalize
                threshold=0.0,
                timestamp=timestamp,
                failure_reason=FailureReason.METHOD_EXCLUDED,
                failure_details="Method is explicitly excluded from calibration system"
            )

        # Get validation threshold
        if override_threshold is not None:
            threshold = override_threshold
        else:
            threshold = self._get_threshold_for_method(
                method_id,
                canonical_method_id=canonical_method_id,
            )

        logger.info(
            "validating_method",
            extra={
                "method_id": method_id,
                "canonical_method_id": canonical_method_id,
                "threshold": threshold,
                "context": context.to_dict() if hasattr(context, 'to_dict') else
```

```
str(context)
            }
        )

        # Perform calibration
        try:
            calibration_result = self.orchestrator.calibrate(
                method_id=canonical_method_id,
                method_version=method_version,
                context=context,
                pdt_structure=pdt_structure,
                graph_config=graph_config,
                subgraph_id=subgraph_id
            )

            final_score = calibration_result.final_score

            # Extract layer scores
            layer_scores = {
                layer_score.layer.value: layer_score.score
                for layer_score in calibration_result.layer_scores.values()
            }

            # Make decision
            if final_score >= threshold:
                decision = ValidationDecision.PASS
                failure_reason = None
                failure_details = None
                recommendations = []
            else:
                decision = ValidationDecision.FAIL
                failure_reason, failure_details = self._analyze_failure(
                    calibration_result, threshold
                )
                recommendations = self._generate_recommendations(
                    calibration_result, failure_reason
                )

            logger.info(
                "validation_complete",
                extra={
                    "method_id": method_id,
                    "canonical_method_id": canonical_method_id,
                    "decision": decision.value,
                    "score": final_score,
                    "threshold": threshold
                }
            )

            return ValidationResult(
                method_id=method_id,
                decision=decision,
                calibration_score=final_score,
                threshold=threshold,
                timestamp=timestamp,
                calibration_result=calibration_result,
                failure_reason=failure_reason,
                failure_details=failure_details,
                layer_scores=layer_scores,
                recommendations=recommendations
            )

        except Exception as e:
            logger.error(
                "validation_error",
                extra={
                    "method_id": method_id,
                    "canonical_method_id": canonical_method_id,
```

```
                "error": str(e),
            },
            exc_info=True
        )

        return ValidationResult(
            method_id=method_id,
            decision=ValidationDecision.FAIL,
            calibration_score=0.0,
            threshold=threshold,
            timestamp=timestamp,
            failure_reason=FailureReason.UNKNOWN,
            failure_details=f"Calibration error: {str(e)}",
            recommendations=["Fix calibration system errors before retrying"]
        )

def validate_plan_executors(
    self,
    plan_id: str,
    context: ContextTuple,
    pdt_structure: PDTStructure,
    executor_names: Optional[List[str]] = None
) -> ValidationReport:
    """
    Validate all executors for a plan.

    Args:
        plan_id: Plan identifier
        context: Execution context
        pdt_structure: PDT structure
        executor_names: Optional list of executor names (defaults to all 30)

    Returns:
        ValidationReport with overall results
    """
    timestamp = datetime.utcnow().isoformat()

    # Default: all 30 executors
    if executor_names is None:
        executor_names = [
            f"D{d}Q{q}_Executor"
            for d in range(1, 7)
            for q in range(1, 6)
        ]

    logger.info(
        "validating_plan_executors",
        extra={
            "plan_id": plan_id,
            "executor_count": len(executor_names)
        }
    )

    # Validate each executor
    results = []
    for executor_name in executor_names:
        result = self.validate_method(
            method_id=executor_name,
            method_version="1.0.0",  # TODO: Get from actual executor
            context=context,
            pdt_structure=pdt_structure,
            subgraph_id=f"{plan_id}_{executor_name}"
        )
        results.append(result)

    # Compute statistics
    passed = sum(1 for r in results if r.decision == ValidationDecision.PASS)
    failed = sum(1 for r in results if r.decision == ValidationDecision.FAIL)
```

```
        conditional = sum(1 for r in results if r.decision ==
ValidationDecision.CONDITIONAL_PASS)
        skipped = sum(1 for r in results if r.decision == ValidationDecision.SKIPPED)

        # Determine overall decision
        total_evaluated = len(executor_names) - skipped
        if total_evaluated == 0:
            overall_decision = ValidationDecision.SKIPPED
            summary = "No executors were evaluated"
        elif failed == 0:
            overall_decision = ValidationDecision.PASS
            summary = f"All {passed} executors passed validation"
        elif passed >= total_evaluated * 0.8:  # 80% threshold
            overall_decision = ValidationDecision.CONDITIONAL_PASS
            summary = f"{passed}/{total_evaluated} executors passed
({passed/total_evaluated:.1%})"
        else:
            overall_decision = ValidationDecision.FAIL
            summary = f"Only {passed}/{total_evaluated} executors passed
({passed/total_evaluated:.1%})"

        report = ValidationReport(
            plan_id=plan_id,
            timestamp=timestamp,
            total_methods=len(executor_names),
            passed=passed,
            failed=failed,
            conditional_pass=conditional,
            skipped=skipped,
            method_results=results,
            overall_decision=overall_decision,
            summary=summary
        )

        logger.info(
            "plan_validation_complete",
            extra={
                "plan_id": plan_id,
                "overall_decision": overall_decision.value,
                "pass_rate": report.pass_rate(),
                "passed": passed,
                "failed": failed
            }
        )

        return report

    def _get_threshold_for_method(
        self,
        method_id: str,
        *,
        canonical_method_id: Optional[str] = None,
    ) -> float:
        """
        Determine the appropriate threshold for a method.

        Args:
            method_id: Method identifier

        Returns:
            Threshold value in [0.0, 1.0]
        """
        # Check if it's an executor
        if "_Executor" in method_id or method_id.startswith("D") and "Q" in method_id:
            threshold = self.parameter_loader.get_executor_threshold(method_id)
            logger.debug(
                "using_executor_threshold",
                extra={"method_id": method_id, "threshold": threshold}
```

```
            )
            return threshold

        # Otherwise, get threshold by role
        canonical = canonical_method_id or method_id
        role = self.intrinsic_loader.get_layer(canonical) if self.intrinsic_loader else
None

        if role:
            threshold = self.parameter_loader.get_validation_threshold_for_role(role)
            logger.debug(
                "using_role_threshold",
                extra={
                    "method_id": method_id,
                    "canonical_method_id": canonical,
                    "role": role,
                    "threshold": threshold,
                }
            )
            return threshold

        # Default: conservative threshold
        default_threshold = 0.70
        logger.debug(
            "using_default_threshold",
            extra={"method_id": method_id, "threshold": default_threshold}
        )
        return default_threshold

    def _resolve_canonical_method_id(self, method_id: str) -> str:
        """
        Resolve display-level identifiers to canonical catalogue identifiers.

        Args:
            method_id: Input identifier (e.g., "D1Q1_Executor")

        Returns:
            Canonical identifier suitable for calibration lookups.
        """
        # Already canonical (src.* namespace)
        if method_id.startswith("src."):
            return method_id

        # If intrinsic loader already knows this identifier, keep it
        if self.intrinsic_loader:
            try:
                if self.intrinsic_loader.get_method_data(method_id):
                    return method_id
            except Exception:
                # Fall back to heuristic resolutions
                pass

        if method_id.endswith("_Executor"):
            canonical = f"src.saaaaaa.core.orchestrator.executors.{method_id}.execute"
            if self.intrinsic_loader:
                try:
                    data = self.intrinsic_loader.get_method_data(canonical)
                    if data:
                        return canonical
                except Exception:
                    # Even if lookup fails, return canonical form for downstream systems
                    pass
            return canonical

        return method_id

    def _analyze_failure(
        self,
```

```
        calibration_result: CalibrationResult,
        threshold: float
    ) -> tuple[FailureReason, str]:
        """
        Analyze why a method failed validation.

        Args:
            calibration_result: Calibration result
            threshold: Threshold that was not met

        Returns:
            Tuple of (FailureReason, detailed_explanation)
        """
        # Find the lowest-scoring layer
        layer_scores = calibration_result.layer_scores
        if not layer_scores:
            return (
                FailureReason.UNKNOWN,
                f"Score {calibration_result.final_score:.3f} < {threshold:.3f}, but no
layer breakdown available"
            )

        lowest_layer = min(layer_scores.values(), key=lambda ls: ls.score)

        # Determine failure reason based on lowest layer
        if lowest_layer.layer == LayerID.BASE:
            return (
                FailureReason.BASE_LAYER_LOW,
                f"Base layer (intrinsic quality) is low: {lowest_layer.score:.3f}. "
                f"Code quality issues detected. {lowest_layer.rationale}"
            )
        elif lowest_layer.layer == LayerID.CHAIN:
            return (
                FailureReason.CHAIN_LAYER_FAIL,
                f"Chain layer (data flow) failed: {lowest_layer.score:.3f}. "
                f"Missing or incompatible inputs. {lowest_layer.rationale}"
            )
        elif lowest_layer.layer == LayerID.CONGRUENCE:
            return (
                FailureReason.CONGRUENCE_FAIL,
                f"Congruence layer (method compatibility) failed:
{lowest_layer.score:.3f}. "
                f"Inconsistent method ensemble. {lowest_layer.rationale}"
            )
        elif lowest_layer.layer == LayerID.UNIT:
            return (
                FailureReason.UNIT_LAYER_FAIL,
                f"Unit layer (PDT quality) is low: {lowest_layer.score:.3f}. "
                f"PDT structure issues detected. {lowest_layer.rationale}"
            )
        elif lowest_layer.layer in {LayerID.QUESTION, LayerID.DIMENSION, LayerID.POLICY}:
            return (
                FailureReason.CONTEXTUAL_FAIL,
                f"Contextual layer ({lowest_layer.layer.value}) incompatible:
{lowest_layer.score:.3f}. "
                f"{lowest_layer.rationale}"
            )
        elif lowest_layer.layer == LayerID.META:
            return (
                FailureReason.META_LAYER_FAIL,
                f"Meta layer (governance) failed: {lowest_layer.score:.3f}. "
                f"Governance/transparency issues. {lowest_layer.rationale}"
            )
        else:
            return (
                FailureReason.SCORE_BELOW_THRESHOLD,
                f"Overall score {calibration_result.final_score:.3f} < {threshold:.3f}. "
                f"Lowest layer: {lowest_layer.layer.value} = {lowest_layer.score:.3f}"
```

```
            )

    def _generate_recommendations(
        self,
        calibration_result: CalibrationResult,
        failure_reason: FailureReason
    ) -> List[str]:
        """
        Generate actionable recommendations based on failure reason.

        Args:
            calibration_result: Calibration result
            failure_reason: Why validation failed

        Returns:
            List of recommendation strings
        """
        recommendations = []

        if failure_reason == FailureReason.BASE_LAYER_LOW:
            recommendations.extend([
                "Improve code quality: add tests, documentation, type hints",
                "Review theoretical foundation of the method",
                "Consider refactoring for better maintainability"
            ])
        elif failure_reason == FailureReason.CHAIN_LAYER_FAIL:
            recommendations.extend([
                "Verify all required inputs are available in execution graph",
                "Check method_signatures.json for correct input specification",
                "Ensure upstream methods are executing correctly"
            ])
        elif failure_reason == FailureReason.CONGRUENCE_FAIL:
            recommendations.extend([
                "Review method ensemble for semantic compatibility",
                "Check method_registry.json for correct semantic tags",
                "Consider using different fusion rules"
            ])
        elif failure_reason == FailureReason.UNIT_LAYER_FAIL:
            recommendations.extend([
                "Improve PDT structure quality",
                "Ensure all mandatory sections are present",
                "Validate indicator and PPI quality"
            ])
        elif failure_reason == FailureReason.CONTEXTUAL_FAIL:
            recommendations.extend([
                "Verify method is appropriate for this question/dimension/policy",
                "Check compatibility registry for correct mappings",
                "Consider using a different method for this context"
            ])
        elif failure_reason == FailureReason.META_LAYER_FAIL:
            recommendations.extend([
                "Improve traceability: export formulas, add logging",
                "Validate governance compliance",
                "Optimize execution time if performance is an issue"
            ])
        else:
            recommendations.append(
                "Review all layer scores to identify specific improvement areas"
            )

        return recommendations

===== FILE: src/saaaaaa/core/calibration/validators.py =====
"""
Three-Pillar Calibration System - Validation Functions

This module implements validation checks for the calibration system
as specified in the SUPERPROMPT Three-Pillar Calibration System.
```

```
Spec compliance: Section 8 (Validation & Governance)
"""

import json
from pathlib import Path
from typing import TYPE_CHECKING, Any

if TYPE_CHECKING:
    from .data_structures import CalibrationCertificate

from .data_structures import REQUIRED_LAYERS, LayerID, MethodRole


class CalibrationValidator:
    """
    Validation system for calibration configs and runtime checks.

    Spec compliance: Section 8 (Validation & Governance)
    """

    def __init__(self, config_dir: str = None) -> None:
        """Initialize validator with config directory"""
        if config_dir is None:
            config_dir = Path(__file__).parent.parent / "config"
        else:
            config_dir = Path(config_dir)

        self.config_dir = config_dir

    def validate_layer_completeness(
        self,
        method_id: str,
        role: MethodRole,
        declared_layers: set[LayerID]
    ) -> tuple[bool, list[str]]:
        """
        Validate that method declares all required layers for its role.

        Spec compliance: Section 4 (Theorem 4.1 - No Silent Defaults)

        Args:
            method_id: Canonical method ID
            role: Method role
            declared_layers: Set of layers method declares

        Returns:
            (is_valid, error_messages)
        """
        required = REQUIRED_LAYERS.get(role, set())
        missing = required - declared_layers

        if not missing:
            return True, []

        errors = [
            f"Method {method_id} (role={role.value}) missing required layers: "
            f"{[l.value for l in missing]}"
        ]
        return False, errors

    def validate_fusion_weights(
        self,
        role_params: dict[str, Any],
        role_name: str
    ) -> tuple[bool, list[str]]:
        """
        Validate fusion weight normalization and constraints.
```

```
        Spec compliance: Section 5 (Fusion Operator Constraints)

        Constraints:
        1. a_ℓ ≥ 0 for all ℓ
        2. a_ℓk ≥ 0 for all (ℓ,k)
        3. Σ(a_ℓ) + Σ(a_ℓk) = 1.0

        Args:
            role_params: Parameters from fusion_specification.json
            role_name: Role identifier

        Returns:
            (is_valid, error_messages)
        """
        errors = []

        linear_weights = role_params.get("linear_weights", {})
        interaction_weights = role_params.get("interaction_weights", {})

        # Check non-negativity
        for layer, weight in linear_weights.items():
            if weight < 0:
                errors.append(
                    f"Role {role_name}: linear weight for {layer} is negative: {weight}"
                )

        for pair, weight in interaction_weights.items():
            if weight < 0:
                errors.append(
                    f"Role {role_name}: interaction weight for {pair} is negative:
{weight}"
                )

        # Check normalization
        linear_sum = sum(linear_weights.values())
        interaction_sum = sum(interaction_weights.values())
        total = linear_sum + interaction_sum

        tolerance = 1e-9
        if abs(total - 1.0) > tolerance:
            errors.append(
                f"Role {role_name}: weights do not sum to 1.0. "
                f"Linear={linear_sum:.6f}, Interaction={interaction_sum:.6f}, "
                f"Total={total:.6f}"
            )

        return len(errors) == 0, errors

    def validate_anti_universality(
        self,
        method_config: dict[str, Any],
        method_id: str,
        contextual_config: dict[str, Any],
        monolith: dict[str, Any]
    ) -> tuple[bool, list[str]]:
        """
        Validate anti-universality constraint.

        Spec compliance: Section 3.4 (Anti-Universality Constraint)

        No method may have maximal compatibility (1.0) with ALL Q, D, and P.

        Args:
            method_config: Method configuration
            method_id: Canonical method ID
            contextual_config: Contextual parametrization config
            monolith: Questionnaire monolith
```

```
        Returns:
            (is_valid, error_messages)
        """
        errors = []

        # For now, ensure policy default is < 1.0 (we set it to 0.9 in config)
        policy_areas = contextual_config.get("layer_policy", {}).get("policy_areas", {})
        all_policies_maximal = True

        for _policy_id, policy_spec in policy_areas.items():
            if policy_spec.get("default_score", 0.0) < 0.99:
                all_policies_maximal = False
                break

        if all_policies_maximal:
            errors.append(
                f"Method {method_id} violates anti-universality: "
                f"all policy areas have maximal (≥0.99) compatibility"
            )

        return len(errors) == 0, errors

    def validate_intrinsic_calibration(
        self,
        config: dict[str, Any]
    ) -> tuple[bool, list[str]]:
        """
        Validate intrinsic_calibration.json structure and values.

        Args:
            config: Loaded intrinsic_calibration.json

        Returns:
            (is_valid, error_messages)
        """
        errors = []

        # Check weights sum to 1.0
        weights = config.get("_base_weights", {})
        weight_sum = weights.get("w_th", 0) + weights.get("w_imp", 0) +
weights.get("w_dep", 0)

        if abs(weight_sum - 1.0) > 1e-9:
            errors.append(f"Base weights do not sum to 1.0: {weight_sum}")

        # Check each method entry
        methods = config.get("methods", {})
        for method_id, method_data in methods.items():
            if method_id.startswith("_"):
                continue  # Skip metadata

            # Check required fields
            for field in ["b_theory", "b_impl", "b_deploy"]:
                if field not in method_data:
                    errors.append(f"Method {method_id} missing field: {field}")
                    continue

                value = method_data[field]
                if not (0.0 <= value <= 1.0):
                    errors.append(
                        f"Method {method_id} field {field} out of bounds: {value}"
                    )

        return len(errors) == 0, errors

    def validate_config_files(self) -> tuple[bool, list[str]]:
        """
```

```
    Validate all three pillar config files.

    Spec compliance: Section 8 (CI / QA Rules)

    This should be run in CI to ensure config integrity.

    Returns:
        (is_valid, error_messages)
    """
    all_errors = []

    # Load configs
    try:
        with open(self.config_dir / "intrinsic_calibration.json") as f:
            intrinsic = json.load(f)
    except Exception as e:
        all_errors.append(f"Failed to load intrinsic_calibration.json: {e}")
        intrinsic = {}

    # Validate contextual config exists (full validation TBD)
    try:
        contextual_path = self.config_dir / "contextual_parametrization.json"
        if not contextual_path.exists():
            all_errors.append("contextual_parametrization.json not found")
    except Exception as e:
        all_errors.append(f"Failed to check contextual_parametrization.json: {e}")

    try:
        with open(self.config_dir / "fusion_specification.json") as f:
            fusion = json.load(f)
    except Exception as e:
        all_errors.append(f"Failed to load fusion_specification.json: {e}")
        fusion = {}

    # Validate intrinsic calibration
    if intrinsic:
        valid, errors = self.validate_intrinsic_calibration(intrinsic)
        all_errors.extend(errors)

    # Validate fusion weights for each role
    if fusion:
        role_params = fusion.get("role_fusion_parameters", {})
        for role_name, params in role_params.items():
            valid, errors = self.validate_fusion_weights(params, role_name)
            all_errors.extend(errors)

    return len(all_errors) == 0, all_errors

def validate_boundedness(
    self,
    layer_scores: dict[str, float],
    calibrated_score: float
) -> tuple[bool, list[str]]:
    """
    Validate boundedness constraint: all scores in [0,1].

    Spec compliance: Section 8 (P1. Boundedness)

    Args:
        layer_scores: All layer scores
        calibrated_score: Final calibrated score

    Returns:
        (is_valid, error_messages)
    """
    errors = []

    # Check layer scores
```

```
        for layer, score in layer_scores.items():
            if not (0.0 <= score <= 1.0):
                errors.append(f"Layer {layer} score out of bounds: {score}")

        # Check calibrated score
        if not (0.0 <= calibrated_score <= 1.0):
            errors.append(f"Calibrated score out of bounds: {calibrated_score}")

        return len(errors) == 0, errors


# Convenience functions
def validate_config_files(config_dir: str = None) -> tuple[bool, list[str]]:
    """
    Validate all calibration config files.

    This should be called in CI/CD pipelines.
    """
    validator = CalibrationValidator(config_dir=config_dir)
    return validator.validate_config_files()


def validate_certificate(
    certificate: 'CalibrationCertificate'
) -> tuple[bool, list[str]]:
    """
    Validate a calibration certificate.

    Args:
        certificate: CalibrationCertificate to validate

    Returns:
        (is_valid, error_messages)
    """
    validator = CalibrationValidator()

    # Check boundedness
    valid, errors = validator.validate_boundedness(
        certificate.layer_scores,
        certificate.calibrated_score
    )

    return valid, errors

===== FILE: src/saaaaaa/core/contracts/runtime_contracts.py =====
"""
Runtime contracts: shared enums and manifest models for fallback tracking.

This module defines the contract types used across the system for tracking
degradations, fallbacks, and runtime behavior. All components use these
shared contracts for consistent observability.
"""

from enum import Enum
from typing import Optional
from pydantic import BaseModel, Field


class LanguageTier(Enum):
    """Language detection result tier indicating detection quality."""

    NORMAL = "normal"
    """Language detected successfully."""

    WARN_DEFAULT_ES = "warn_default_es"
    """Detection failed with LangDetectException, defaulted to Spanish."""

    FAIL = "fail"
```

```
    """Detection failed with unexpected error."""


class SegmentationMethod(Enum):
    """Text segmentation method used for document processing."""

    SPACY_LG = "spacy_lg"
    """spaCy large model (es_core_news_lg)."""

    SPACY_MD = "spacy_md"
    """spaCy medium model (es_core_news_md)."""

    SPACY_SM = "spacy_sm"
    """spaCy small model (es_core_news_sm)."""

    REGEX = "regex"
    """Regex-based segmentation fallback."""

    LINE = "line"
    """Line-based segmentation fallback."""


class CalibrationMode(Enum):
    """Calibration completeness mode."""

    FULL = "full"
    """Complete calibration with all required fields."""

    DEFAULTED = "defaulted"
    """Some calibration values defaulted."""

    PARTIAL = "partial"
    """Partial calibration with missing optional fields."""


class DocumentIdSource(Enum):
    """Source of document identifier."""

    METADATA = "metadata"
    """Document ID from metadata (preferred)."""

    FILENAME_FALLBACK = "filename_fallback"
    """Document ID from filename (fallback)."""


class ExecutionMetricsMode(Enum):
    """Execution metrics collection mode."""

    REAL = "real"
    """Real execution metrics collected."""

    ESTIMATED = "estimated"
    """Execution metrics estimated (fallback)."""


class FallbackCategory(Enum):
    """
    Fallback categorization for SLO tracking.

    Categories:
        A: Critical - never allowed in PROD (e.g., missing core modules)
        B: Quality degradation - SLO < 5% in PROD (e.g., regex segmentation)
        C: Dev/bootstrap only - not available in PROD
        D: Migration path - tracked for reduction (e.g., filename fallback)
    """

    A = "A"
    """Critical fallback - never allowed in PROD."""
```

```
    B = "B"
    """Quality degradation - SLO < 5% in PROD."""

    C = "C"
    """Development/bootstrap only."""

    D = "D"
    """Migration path - tracked for reduction."""


# Manifest Models

class LanguageDetectionInfo(BaseModel):
    """Language detection result information for manifest."""

    tier: LanguageTier = Field(..., description="Detection result tier")
    detected_language: Optional[str] = Field(None, description="Detected language code
(e.g., 'es', 'en')")
    reason: Optional[str] = Field(None, description="Reason for fallback if tier !=
NORMAL")

    class Config:
        frozen = True


class SegmentationInfo(BaseModel):
    """Text segmentation method information for manifest."""

    method: SegmentationMethod = Field(..., description="Segmentation method used")
    downgraded_from: Optional[SegmentationMethod] = Field(None, description="Original
method if downgraded")
    reason: Optional[str] = Field(None, description="Reason for downgrade")

    class Config:
        frozen = True


class CalibrationInfo(BaseModel):
    """Calibration completeness information for manifest."""

    mode: CalibrationMode = Field(..., description="Calibration mode")
    defaulted_count: int = Field(0, description="Number of defaulted values")
    partial_count: int = Field(0, description="Number of partial calibrations")
    missing_base_weights: bool = Field(False, description="Whether _base_weights is
missing")

    class Config:
        frozen = True


class ExecutionMetricsInfo(BaseModel):
    """Execution metrics collection information for manifest."""

    mode: ExecutionMetricsMode = Field(..., description="Metrics collection mode")
    estimated_phases: list[str] = Field(default_factory=list, description="Phases with
estimated metrics")
    reason: Optional[str] = Field(None, description="Reason for estimation")

    class Config:
        frozen = True


class DocumentIdInfo(BaseModel):
    """Document identifier source information for manifest."""

    source: DocumentIdSource = Field(..., description="Source of document ID")
    document_id: str = Field(..., description="Actual document ID used")
```

```
    fallback_reason: Optional[str] = Field(None, description="Reason for fallback if
source != METADATA")

    class Config:
        frozen = True


class ContradictionInfo(BaseModel):
    """Contradiction detection mode information for manifest."""

    mode: str = Field(..., description="Detection mode: 'full' or 'fallback'")
    module_available: bool = Field(..., description="Whether contradiction module is
available")
    reason: Optional[str] = Field(None, description="Reason for fallback if mode ==
'fallback'")

    class Config:
        frozen = True


class GraphMetricsInfo(BaseModel):
    """Graph metrics computation information for manifest."""

    computed: bool = Field(..., description="Whether graph metrics were computed")
    networkx_available: bool = Field(..., description="Whether NetworkX is available")
    reason: Optional[str] = Field(None, description="Reason for skip if not computed")

    class Config:
        frozen = True


class RuntimeManifest(BaseModel):
    """
    Complete runtime manifest tracking all degradations and fallbacks.

    This manifest is included in pipeline outputs to provide complete
    observability of runtime behavior and degradations.
    """

    runtime_mode: str = Field(..., description="Runtime mode (prod/dev/exploratory)")
    language_detection: Optional[LanguageDetectionInfo] = None
    segmentation: Optional[SegmentationInfo] = None
    calibration: Optional[CalibrationInfo] = None
    execution_metrics: Optional[ExecutionMetricsInfo] = None
    document_id: Optional[DocumentIdInfo] = None
    contradiction: Optional[ContradictionInfo] = None
    graph_metrics: Optional[GraphMetricsInfo] = None

    class Config:
        frozen = True

===== FILE: src/saaaaaa/core/contracts.py =====
"""Core-layer contract aliases.

This module provides stable import paths for TypedDict contracts used across
core orchestration code.  The actual contract definitions live in
``saaaaaa.utils.core_contracts`` which centralises documentation and runtime
validation.  Re-exporting them here keeps core modules decoupled from the
utilities package and prevents circular imports when infrastructure code wants
to consume the same contracts.
"""

from saaaaaa.utils.core_contracts import (
    CDAFFrameworkInputContract,
    CDAFFrameworkOutputContract,
    ContradictionDetectorInputContract,
    ContradictionDetectorOutputContract,
    DocumentData,
```

```
    EmbeddingPolicyInputContract,
    EmbeddingPolicyOutputContract,
    PDETAnalyzerInputContract,
    PDETAnalyzerOutputContract,
    PolicyProcessorInputContract,
    PolicyProcessorOutputContract,
    SemanticAnalyzerInputContract,
    SemanticAnalyzerOutputContract,
    SemanticChunkingInputContract,
    SemanticChunkingOutputContract,
    TeoriaCambioInputContract,
    TeoriaCambioOutputContract,
)

__all__ = [
    "CDAFFrameworkInputContract",
    "CDAFFrameworkOutputContract",
    "ContradictionDetectorInputContract",
    "ContradictionDetectorOutputContract",
    "DocumentData",
    "EmbeddingPolicyInputContract",
    "EmbeddingPolicyOutputContract",
    "PDETAnalyzerInputContract",
    "PDETAnalyzerOutputContract",
    "PolicyProcessorInputContract",
    "PolicyProcessorOutputContract",
    "SemanticAnalyzerInputContract",
    "SemanticAnalyzerOutputContract",
    "SemanticChunkingInputContract",
    "SemanticChunkingOutputContract",
    "TeoriaCambioInputContract",
    "TeoriaCambioOutputContract",
]

===== FILE: src/saaaaaa/core/dependency_lockdown.py =====
"""Dependency lockdown enforcement to prevent magic downloads and hidden behavior.

This module enforces explicit dependency management by:
1. Checking if online model downloads are allowed via HF_ONLINE env var
2. Setting HuggingFace offline mode when online access is disabled
3. Providing early failure for missing critical dependencies
4. Allowing explicit degraded mode marking for optional dependencies

No fallback logic, no "best effort" embeddings. Either dependencies are present
and configured correctly, or the system fails fast with clear error messages.
"""

import logging
import os

logger = logging.getLogger(__name__)


def _is_model_cached(model_name: str) -> bool:
    """Check if a HuggingFace model is cached locally.

    Uses a heuristic check of common cache locations to determine if a model
    is likely available offline. This is a best-effort check - false positives
    are acceptable (will fail later when model actually loads), but false negatives
    should be minimized to avoid blocking offline usage of cached models.

    Args:
        model_name: HuggingFace model name (e.g., "sentence-transformers/model")

    Returns:
        True if model appears to be cached locally, False otherwise
    """
    from pathlib import Path
```

```
    # Check common HuggingFace cache locations
    cache_dirs = [
        os.path.expanduser("~/.cache/huggingface/hub"),
        os.path.expanduser("~/.cache/torch/sentence_transformers"),
        os.getenv("HF_HOME"),
        os.getenv("TRANSFORMERS_CACHE"),
    ]

    # Convert model name to cache directory pattern
    # HF uses "models--org--name" format in cache
    model_slug = model_name.replace("/", "--")

    for cache_dir in cache_dirs:
        if cache_dir and os.path.exists(cache_dir):
            cache_path = Path(cache_dir)
            # Use glob with specific pattern instead of rglob for efficiency
            # Check just the top-level directories, not recursive
            if any(model_slug in p.name for p in cache_path.iterdir()):
                return True

    return False


class DependencyLockdownError(RuntimeError):
    """Raised when a dependency constraint is violated."""
    pass


class DependencyLockdown:
    """Enforces strict dependency controls to prevent hidden/magic behavior.

    This class ensures that:
    - Online model downloads are explicitly controlled via HF_ONLINE env var
    - HuggingFace models only download when explicitly allowed
    - Critical dependencies fail fast if missing
    - Optional dependencies are clearly marked as degraded when missing
    """

    def __init__(self) -> None:
        """Initialize dependency lockdown based on environment configuration."""
        self.hf_allowed = os.getenv("HF_ONLINE", "0") == "1"
        self._enforce_offline_mode()
        self._log_configuration()

    def _enforce_offline_mode(self) -> None:
        """Enforce HuggingFace offline mode if HF_ONLINE is not enabled."""
        if not self.hf_allowed:
            # Set HuggingFace environment variables to prevent downloads
            os.environ["HF_HUB_OFFLINE"] = "1"
            os.environ["TRANSFORMERS_OFFLINE"] = "1"
            logger.info(
                "Dependency lockdown: HuggingFace offline mode ENFORCED "
                "(HF_ONLINE=0 or not set)"
            )
        else:
            logger.warning(
                "Dependency lockdown: HuggingFace online mode ENABLED "
                "(HF_ONLINE=1). Models may be downloaded from HuggingFace Hub."
            )

    def _log_configuration(self) -> None:
        """Log current dependency lockdown configuration."""
        logger.info(
            f"Dependency lockdown initialized: "
            f"HF_ONLINE={self.hf_allowed}, "
            f"HF_HUB_OFFLINE={os.getenv('HF_HUB_OFFLINE', 'unset')}, "
            f"TRANSFORMERS_OFFLINE={os.getenv('TRANSFORMERS_OFFLINE', 'unset')}"
```

```
        )

    def check_online_model_access(
        self,
        model_name: str,
        operation: str = "model download"
    ) -> None:
        """Check if online model access is allowed, raise if not.

        Args:
            model_name: Name of the model being accessed
            operation: Description of the operation (for error message)

        Raises:
            DependencyLockdownError: If online access is not allowed
        """
        if not self.hf_allowed:
            raise DependencyLockdownError(
                f"Online model download disabled in this environment. "
                f"Attempted operation: {operation} for model '{model_name}'. "
                f"To enable online downloads, set HF_ONLINE=1 environment variable. "
                f"No fallback to degraded mode - this is a hard failure."
            )

    def check_critical_dependency(
        self,
        module_name: str,
        pip_package: str,
        phase: str | None = None
    ) -> None:
        """Check if a critical dependency is available, fail fast if not.

        Args:
            module_name: Python module name to import
            pip_package: pip package name for installation instructions
            phase: Optional phase name where dependency is required

        Raises:
            DependencyLockdownError: If critical dependency is missing
        """
        try:
            __import__(module_name)
        except ImportError as e:
            phase_info = f" for phase '{phase}'" if phase else ""
            raise DependencyLockdownError(
                f"Critical dependency '{module_name}' is missing{phase_info}. "
                f"Install it with: pip install {pip_package}. "
                f"No degraded mode available - this is a mandatory dependency. "
                f"Original error: {e}"
            ) from e

    def check_optional_dependency(
        self,
        module_name: str,
        pip_package: str,
        feature: str
    ) -> bool:
        """Check if an optional dependency is available.

        Args:
            module_name: Python module name to import
            pip_package: pip package name for installation instructions
            feature: Feature name that requires this dependency

        Returns:
            True if dependency is available, False otherwise

        Note:
```

```
        This does NOT raise an error, but logs a warning about degraded mode.
        Caller must explicitly handle degraded mode and log it clearly.
        """
        try:
            __import__(module_name)
            return True
        except ImportError:
            logger.warning(
                f"DEGRADED MODE: Optional dependency '{module_name}' not available. "
                f"Feature '{feature}' will be disabled. "
                f"Install with: pip install {pip_package}"
            )
            return False

    def get_mode_description(self) -> dict[str, str | bool]:
        """Get current dependency lockdown mode description.

        Returns:
            Dictionary with mode information for logging/debugging
        """
        return {
            "hf_online_allowed": self.hf_allowed,
            "hf_hub_offline": os.getenv("HF_HUB_OFFLINE", "unset"),
            "transformers_offline": os.getenv("TRANSFORMERS_OFFLINE", "unset"),
            "mode": "online" if self.hf_allowed else "offline_enforced",
        }


# Global singleton instance
_lockdown_instance: DependencyLockdown | None = None


def get_dependency_lockdown() -> DependencyLockdown:
    """Get or create the global dependency lockdown instance.

    Returns:
        Global DependencyLockdown instance
    """
    global _lockdown_instance
    if _lockdown_instance is None:
        _lockdown_instance = DependencyLockdown()
    return _lockdown_instance


def reset_dependency_lockdown() -> None:
    """Reset the global dependency lockdown instance (for testing)."""
    global _lockdown_instance
    _lockdown_instance = None

===== FILE: src/saaaaaa/core/layer_coexistence.py =====
"""
Layer Coexistence and Influence Framework

Formal mathematical framework for method calibration based on layer-specific
evidence aggregation with theoretically grounded fusion operators.

Theoretical Foundations:
- Bayesian inference (Pearl, 1988; Jaynes, 2003)
- Multi-criteria decision analysis (Keeney & Raiffa, 1976)
- Policy coherence structures (Nilsson et al., 2012)
- Ensemble learning (Dietterich, 2000; Wolpert, 1992)
- Fuzzy measures and aggregation (Yager, 1988; Beliakov et al., 2007)

Canonical Layer Notation (from questionnaire_monolith.json):
- @q: Question Layer (300 questions: Q001-Q300)
- @d: Dimension Layer (6 dimensions: DIM01-DIM06)
- @p: Policy Area Layer (10 areas: PA01-PA10)
- @C: Congruence Layer (ensemble compatibility)
```

```
- @m: Meta Layer (cross-layer aggregation)
"""

from dataclasses import dataclass, field
from enum import Enum


class Layer(Enum):
    """
    Canonical layer identifiers.

    Source: questionnaire_monolith.json and theoretical framework specification.
    These are the ONLY valid layer identifiers in the system.
    """
    QUESTION = "@q"      # Evidence-weighted Bayesian scoring
    DIMENSION = "@d"     # Multi-criteria value functions
    POLICY_AREA = "@p"   # Policy coherence structures
    CONGRUENCE = "@C"    # Ensemble compatibility
    META = "@m"          # Generalized aggregation


@dataclass(frozen=True)
class LayerScore:
    """
    Score from a single layer for a method.

    Attributes:
        layer: The layer identifier
        value: Numerical score in [0, 1]
        weight: Importance weight in [0, 1]
        metadata: Additional layer-specific information
    """
    layer: Layer
    value: float
    weight: float = 1.0
    metadata: dict = field(default_factory=dict)

    def __new__(cls, layer: Layer, value: float, weight: float = 1.0, metadata: dict =
None):
        """Validate score bounds before instance creation"""
        if not 0 <= value <= 1:
            raise ValueError(f"Layer score must be in [0,1], got {value}")
        if not 0 <= weight <= 1:
            raise ValueError(f"Layer weight must be in [0,1], got {weight}")
        if metadata is None:
            metadata = {}
        return super().__new__(cls)

    def __init__(self, layer: Layer, value: float, weight: float = 1.0, metadata: dict =
None) -> None:
        # dataclass will set fields, but we need to ensure metadata is not None
        if metadata is None:
            object.__setattr__(self, 'metadata', {})
@dataclass
class MethodSignature:
    """
    Complete signature for a method under Layer Coexistence framework.

    This is the canonical method notation that every calibrated method must expose.

    Attributes:
        method_id: Unique identifier (ClassName.method_name)
        active_layers: Set of layers relevant to this method (L(M))
        input_schema: Dict describing required inputs
        output_schema: Dict describing output space
        fusion_operator_name: Name of the fusion operator F_M
        fusion_parameters: Parameters for F_M
        calibration_rule: Human-readable calibration rule description
```

```
    """
    method_id: str
    active_layers: set[Layer]
    input_schema: dict
    output_schema: dict
    fusion_operator_name: str
    fusion_parameters: dict
    calibration_rule: str

    def to_dict(self) -> dict:
        """Export to dictionary for serialization"""
        return {
            'method_id': self.method_id,
            'active_layers': [layer.value for layer in self.active_layers],
            'input_schema': self.input_schema,
            'output_schema': self.output_schema,
            'fusion_operator_name': self.fusion_operator_name,
            'fusion_parameters': self.fusion_parameters,
            'calibration_rule': self.calibration_rule
        }

    @classmethod
    def from_dict(cls, data: dict) -> 'MethodSignature':
        """Load from dictionary"""
        return cls(
            method_id=data['method_id'],
            active_layers={Layer(layer_str) for layer_str in data['active_layers']},
            input_schema=data['input_schema'],
            output_schema=data['output_schema'],
            fusion_operator_name=data['fusion_operator_name'],
            fusion_parameters=data['fusion_parameters'],
            calibration_rule=data['calibration_rule']
        )


class FusionOperator:
    """
    Abstract base class for fusion operators F_M.

    All fusion operators must satisfy:
    - Monotonicity: ∂F/∂x_ℓ ≥ 0 for all layers ℓ
    - Boundedness: F: [0,1]^n → [0,1]
    - Interpretability: Clear semantic meaning of output

    Subclasses must implement:
    - fuse(scores: List[LayerScore]) -> float
    - verify_properties() -> Dict[str, bool]
    - get_formula() -> str
    """

    def __init__(self, name: str, parameters: dict) -> None:
        self.name = name
        self.parameters = parameters

    def fuse(self, scores: list[LayerScore]) -> float:
        """
        Aggregate layer scores into calibrated output.

        Args:
            scores: List of LayerScore objects

        Returns:
            Calibrated score in [0, 1]
        """
        raise NotImplementedError("Subclasses must implement fuse()")

    def verify_properties(self) -> dict[str, bool]:
        """
```

```
        Verify mathematical properties (monotonicity, boundedness, etc.)

        Returns:
            Dict mapping property name to verification result
        """
        raise NotImplementedError("Subclasses must implement verify_properties()")

    def get_formula(self) -> str:
        """
        Return explicit mathematical formula in canonical notation.

        Returns:
            LaTeX-style formula string
        """
        raise NotImplementedError("Subclasses must implement get_formula()")

    def get_trace(self, scores: list[LayerScore]) -> list[str]:
        """
        Generate step-by-step arithmetic trace.

        Args:
            scores: List of LayerScore objects

        Returns:
            List of computation steps as strings
        """
        raise NotImplementedError("Subclasses must implement get_trace()")


class WeightedAverageFusion(FusionOperator):
    """
    Weighted average fusion operator.

    Formula: F_M(x) = Σ(w_ℓ · x_ℓ) / Σ(w_ℓ)

    Properties:
    - Monotonic: Yes (∂F/∂x_ℓ = w_ℓ/Σw > 0)
    - Bounded: Yes (min(x) ≤ F ≤ max(x))
    - Idempotent: Yes (F(c, c, ..., c) = c)
    - Compensatory: Full (low scores can be compensated by high scores)

    Reference: Standard weighted mean in MCDA (Keeney & Raiffa, 1976, Ch. 3)
    """

    def __init__(self, parameters: dict | None = None) -> None:
        super().__init__("WeightedAverage", parameters or {})
        self.normalize_weights = parameters.get('normalize_weights', True) if parameters
else True

    def fuse(self, scores: list[LayerScore]) -> float:
        """Compute weighted average"""
        if not scores:
            return 0.0

        weighted_sum = sum(score.value * score.weight for score in scores)
        weight_sum = sum(score.weight for score in scores)

        if weight_sum == 0:
            return 0.0

        return weighted_sum / weight_sum

    def verify_properties(self) -> dict[str, bool]:
        """Verify mathematical properties"""
        # Test monotonicity with sample inputs
        test_scores_low = [LayerScore(Layer.QUESTION, 0.3, 1.0)]
        test_scores_high = [LayerScore(Layer.QUESTION, 0.7, 1.0)]
```

```
        result_low = self.fuse(test_scores_low)
        result_high = self.fuse(test_scores_high)

        return {
            'monotonic': result_high >= result_low,
            'bounded': 0 <= result_low <= 1 and 0 <= result_high <= 1,
            'idempotent': abs(self.fuse([LayerScore(Layer.QUESTION, 0.5, 1.0)]) - 0.5) <
1e-10
        }

    def get_formula(self) -> str:
        """Return LaTeX formula"""
        return r"F_{WA}(x) = \frac{\sum_{\ell \in L(M)} w_\ell \cdot x_\ell}{\sum_{\ell
\in L(M)} w_\ell}"

    def get_trace(self, scores: list[LayerScore]) -> list[str]:
        """Generate computation trace"""
        trace = []
        trace.append(f"Weighted Average Fusion: {len(scores)} layers")

        for _i, score in enumerate(scores):
            trace.append(f"  Layer {score.layer.value}: x = {score.value:.4f}, w =
{score.weight:.4f}")

        weighted_sum = sum(s.value * s.weight for s in scores)
        weight_sum = sum(s.weight for s in scores)

        trace.append(f"Weighted sum: Σ(w·x) = {weighted_sum:.4f}")
        trace.append(f"Weight sum: Σ(w) = {weight_sum:.4f}")
        if weight_sum == 0:
            trace.append("Result: No valid weights, returning 0.0")
        else:
            trace.append(f"Result: {weighted_sum:.4f} / {weight_sum:.4f} =
{weighted_sum/weight_sum:.4f}")

        return trace


class OWAFusion(FusionOperator):
    """
    Ordered Weighted Averaging (OWA) fusion operator.

    Formula: F_OWA(x) = Σ(v_i · x_(i))
    where x_(i) is the i-th largest value and v_i are position weights

    Properties:
    - Monotonic: Yes (if all v_i ≥ 0)
    - Bounded: Yes
    - Allows modeling of optimism/pessimism (andness/orness)

    Reference: Yager (1988) "On ordered weighted averaging aggregation operators"
               Int. J. General Systems, 14(3), 183-194

    Parameters:
        weights: Position-based weights [v_1, v_2, ..., v_n]
        Should sum to 1 for proper normalization
    """

    def __init__(self, parameters: dict) -> None:
        super().__init__("OWA", parameters)
        self.position_weights = parameters.get('weights', [])

        if len(self.position_weights) == 0:
            raise ValueError("OWA requires position weights")

    def fuse(self, scores: list[LayerScore]) -> float:
        """Compute OWA aggregation"""
        if not scores:
```

```
            return 0.0

        # Sort scores in descending order
        values = [score.value for score in scores]
        sorted_values = sorted(values, reverse=True)

        # Pad or truncate weights if necessary
        n = len(sorted_values)
        if len(self.position_weights) < n:
            # Extend with equal weights
            extended_weights = list(self.position_weights) + [1.0/n] * (n -
len(self.position_weights))
        else:
            extended_weights = self.position_weights[:n]

        # Normalize weights
        weight_sum = sum(extended_weights)
        if weight_sum > 0:
            extended_weights = [w / weight_sum for w in extended_weights]

        # Compute weighted sum
        result = sum(w * v for w, v in zip(extended_weights, sorted_values, strict=False))
        return float(result)

    def verify_properties(self) -> dict[str, bool]:
        """Verify OWA properties"""
        # Check monotonicity, boundedness
        test_scores = [
            LayerScore(Layer.QUESTION, 0.2, 1.0),
            LayerScore(Layer.DIMENSION, 0.5, 1.0),
            LayerScore(Layer.POLICY_AREA, 0.8, 1.0)
        ]

        result = self.fuse(test_scores)
        weight_sum = sum(self.position_weights)

        return {
            'monotonic': True,  # Always true if weights are non-negative
            'bounded': 0 <= result <= 1,
            'weights_sum_to_one': abs(weight_sum - 1.0) < 1e-6
        }

    def get_formula(self) -> str:
        """Return LaTeX formula"""
        return r"F_{OWA}(x) = \sum_{i=1}^{n} v_i \cdot x_{(i)} \text{ where } x_{(i)}
\text{ is i-th largest}"

    def get_trace(self, scores: list[LayerScore]) -> list[str]:
        """Generate computation trace"""
        trace = []
        trace.append(f"OWA Fusion: {len(scores)} layers")

        values = [(score.layer.value, score.value) for score in scores]
        values_sorted = sorted(values, key=lambda x: x[1], reverse=True)

        trace.append("Sorted values (descending):")
        for i, (layer, val) in enumerate(values_sorted):
            weight_idx = min(i, len(self.position_weights) - 1)
            weight = self.position_weights[weight_idx]
            trace.append(f"  Position {i+1}: {layer} = {val:.4f}, weight = {weight:.4f}")

        result = self.fuse(scores)
        trace.append(f"Result: {result:.4f}")

        return trace


# Registry of available fusion operators
```

```
FUSION_OPERATORS = {
    'WeightedAverage': WeightedAverageFusion,
    'OWA': OWAFusion,
}


def create_fusion_operator(name: str, parameters: dict | None = None) -> FusionOperator:
    """
    Factory function to create fusion operators.

    Args:
        name: Operator name from FUSION_OPERATORS
        parameters: Operator-specific parameters

    Returns:
        Configured FusionOperator instance
    """
    if name not in FUSION_OPERATORS:
        raise ValueError(f"Unknown fusion operator: {name}. Available:
{list(FUSION_OPERATORS.keys())}")

    operator_class = FUSION_OPERATORS[name]
    return operator_class(parameters or {})

===== FILE: src/saaaaaa/core/layer_influence_model.py =====
"""
Layer Coexistence and Influence Model - Formal Specification

This module encodes the mathematical relationships between layers,
including:
- Conditional activation rules (when a layer becomes relevant)
- Influence relationships (how layers weight/transform each other)
- Coexistence constraints (compatibility requirements)

All rules are explicit, verifiable, and derived from theoretical foundations.

References:
- Pearl (1988): Probabilistic Reasoning in Intelligent Systems (conditional independence)
- Keeney & Raiffa (1976): Decisions with Multiple Objectives (preference independence)
- Grabisch (1997): k-order additive discrete fuzzy measures (interaction indices)
"""

from collections.abc import Callable
from dataclasses import dataclass, field
from enum import Enum

from .layer_coexistence import Layer, LayerScore


class LayerInfluenceType(Enum):
    """Types of influence one layer can have on another."""
    WEIGHTING = "weighting"          # Layer A modifies weight of Layer B
    TRANSFORMATION = "transformation"  # Layer A transforms values from Layer B
    ACTIVATION = "activation"         # Layer A determines if Layer B is active
    CONSTRAINT = "constraint"         # Layer A constrains valid values of Layer B


@dataclass
class LayerInfluence:
    """
    Formal specification of how one layer influences another.

    Attributes:
        source_layer: The influencing layer
        target_layer: The influenced layer
        influence_type: Type of influence relationship
        strength: Strength of influence in [0, 1]
        functional_form: Mathematical description of influence
```

```
        conditions: When this influence applies
    """
    source_layer: Layer
    target_layer: Layer
    influence_type: LayerInfluenceType
    strength: float
    functional_form: str
    conditions: dict = field(default_factory=dict)

    def __post_init__(self):
        """Validate influence specification"""
        if not 0 <= self.strength <= 1:
            raise ValueError(f"Influence strength must be in [0,1], got {self.strength}")
        if self.source_layer == self.target_layer:
            raise ValueError("Self-influence not permitted in current model")


@dataclass
class LayerActivationRule:
    """
    Rule determining when a layer becomes active for a method.

    This encodes the endogenous determination of L(M) from method characteristics.

    Attributes:
        layer: The layer this rule applies to
        triggers: Conditions that activate this layer
        prerequisites: Other layers that must be active first
        priority: Activation priority (higher = checked first)
    """
    layer: Layer
    triggers: list[Callable]  # Functions that return bool
    prerequisites: set[Layer] = field(default_factory=set)
    priority: int = 0

    def check_activation(self, method_characteristics: dict) -> bool:
        """
        Check if this layer should be active given method characteristics.

        Args:
            method_characteristics: Dict describing method properties

        Returns:
            True if layer should be active
        """
        return any(trigger(method_characteristics) for trigger in self.triggers)


class LayerCoexistenceModel:
    """
    Formal model of layer interactions and dependencies.

    This encodes the complete "Layer Coexistence and Influence" system,
    including:
    - How to determine L(M) from method properties
    - How layers influence each other
    - Valid coexistence patterns
    - Composition rules for multi-layer fusion

    Design Principle: All layer interactions are explicit, not implicit.
    No hidden dependencies or undocumented couplings permitted.
    """

    def __init__(self) -> None:
        self.influences: list[LayerInfluence] = []
        self.activation_rules: dict[Layer, LayerActivationRule] = {}
        self.compatibility_matrix: dict[tuple[Layer, Layer], float] = {}
```

```
        # Initialize canonical layer relationships
        self._initialize_canonical_relationships()

    def _initialize_canonical_relationships(self) -> None:
        """
        Define canonical layer relationships based on theoretical model.

        Canonical Relationships:

        1. @q → @d (WEIGHTING): Question-level evidence weights dimension scores
           - High certainty at @q increases weight of @d
           - Functional form: w_d' = w_d · (1 + α·certainty_q)

        2. @d → @p (ACTIVATION): Dimensions determine relevant policy areas
           - If dimension scores exist, policy coherence becomes relevant
           - Functional form: active(@p) ⟺ |scored_dimensions| ≥ threshold

        3. @p → @C (CONSTRAINT): Policy areas constrain ensemble methods
           - Policy structure limits valid ensemble combinations
           - Functional form: valid_ensembles ⊆ compatible_with_policy_structure

        4. {@q, @d, @p} → @m (TRANSFORMATION): Base layers feed meta-aggregation
           - Meta layer synthesizes across evidence levels
           - Functional form: x_m = g(x_q, x_d, x_p) where g is aggregation

        5. @C → @m (WEIGHTING): Congruence modulates meta-layer confidence
           - Ensemble agreement increases meta-layer weight
           - Functional form: w_m' = w_m · congruence_score
        """

        # @q → @d influence
        self.add_influence(LayerInfluence(
            source_layer=Layer.QUESTION,
            target_layer=Layer.DIMENSION,
            influence_type=LayerInfluenceType.WEIGHTING,
            strength=0.5,
            functional_form="w_d' = w_d * (1 + 0.5 * certainty_q)",
            conditions={'requires': 'question_certainty_available'}
        ))

        # @d → @p influence
        self.add_influence(LayerInfluence(
            source_layer=Layer.DIMENSION,
            target_layer=Layer.POLICY_AREA,
            influence_type=LayerInfluenceType.ACTIVATION,
            strength=1.0,
            functional_form="active(@p) ⟺ |scored_dimensions| ≥ 3",
            conditions={'threshold': 3}
        ))

        # @p → @C influence
        self.add_influence(LayerInfluence(
            source_layer=Layer.POLICY_AREA,
            target_layer=Layer.CONGRUENCE,
            influence_type=LayerInfluenceType.CONSTRAINT,
            strength=0.7,
            functional_form="valid_ensembles ⊆ policy_compatible_ensembles",
            conditions={'requires': 'policy_structure_defined'}
        ))

        # Base layers → @m influence
        for base_layer in [Layer.QUESTION, Layer.DIMENSION, Layer.POLICY_AREA]:
            self.add_influence(LayerInfluence(
                source_layer=base_layer,
                target_layer=Layer.META,
                influence_type=LayerInfluenceType.TRANSFORMATION,
                strength=0.33,
                functional_form="x_m = weighted_mean(x_q, x_d, x_p)",
```

```
            conditions={'aggregation_type': 'weighted_mean'}
        ))

    # @C → @m influence
    self.add_influence(LayerInfluence(
        source_layer=Layer.CONGRUENCE,
        target_layer=Layer.META,
        influence_type=LayerInfluenceType.WEIGHTING,
        strength=0.6,
        functional_form="w_m' = w_m * congruence_score",
        conditions={'requires': 'ensemble_agreement'}
    ))

    # Initialize compatibility matrix (all pairs compatible by default)
    for layer1 in Layer:
        for layer2 in Layer:
            # Diagonal: perfect self-compatibility
            if layer1 == layer2:
                self.compatibility_matrix[(layer1, layer2)] = 1.0
            # Off-diagonal: initialize to compatible
            else:
                self.compatibility_matrix[(layer1, layer2)] = 0.8

    # Adjust specific incompatibilities if any
    # (Currently all layers are mutually compatible)

def add_influence(self, influence: LayerInfluence) -> None:
    """Register a layer influence relationship."""
    self.influences.append(influence)

def add_activation_rule(self, rule: LayerActivationRule) -> None:
    """Register a layer activation rule."""
    self.activation_rules[rule.layer] = rule

def determine_active_layers(
    self,
    method_characteristics: dict
) -> set[Layer]:
    """
    Determine L(M) endogenously from method characteristics.

    This is the key function that derives active layers from method
    properties rather than requiring manual specification.

    Args:
        method_characteristics: Dict with keys like:
            - 'operates_on_questions': bool
            - 'aggregates_dimensions': bool
            - 'addresses_policy_areas': bool
            - 'uses_ensemble': bool
            - 'performs_meta_aggregation': bool
            - 'question_count': int
            - 'dimension_count': int
            - 'policy_area_count': int

    Returns:
        Set of active layers for this method
    """
    active = set()

    # Sort rules by priority
    sorted_rules = sorted(
        self.activation_rules.items(),
        key=lambda x: x[1].priority,
        reverse=True
    )

    # Check each rule
```

```
    for layer, rule in sorted_rules:
        # Check prerequisites
        if not rule.prerequisites.issubset(active):
            continue

        # Check triggers
        if rule.check_activation(method_characteristics):
            active.add(layer)

    return active

def get_layer_influences(
    self,
    target_layer: Layer,
    active_layers: set[Layer]
) -> list[LayerInfluence]:
    """
    Get all influences affecting a target layer from active layers.

    Args:
        target_layer: Layer being influenced
        active_layers: Set of currently active layers

    Returns:
        List of applicable LayerInfluence objects
    """
    return [
        inf for inf in self.influences
        if inf.target_layer == target_layer
        and inf.source_layer in active_layers
    ]

def compute_effective_weight(
    self,
    target_layer: Layer,
    base_weight: float,
    layer_scores: dict[Layer, LayerScore],
    active_layers: set[Layer]
) -> float:
    """
    Compute effective weight for a layer after applying influences.

    Args:
        target_layer: Layer whose weight is being computed
        base_weight: Initial weight
        layer_scores: Scores for all active layers
        active_layers: Set of active layers

    Returns:
        Effective weight after influence application
    """
    effective_weight = base_weight

    # Get weighting influences
    influences = [
        inf for inf in self.get_layer_influences(target_layer, active_layers)
        if inf.influence_type == LayerInfluenceType.WEIGHTING
    ]

    for influence in influences:
        source_score = layer_scores.get(influence.source_layer)
        if source_score:
            # Apply influence (simplified model)
            modifier = 1.0 + influence.strength * (source_score.value - 0.5)
            effective_weight *= modifier

    # Ensure weight stays in valid range
    return max(0.0, min(1.0, effective_weight))
```

```
    def check_compatibility(
        self,
        layer_set: set[Layer]
    ) -> tuple[bool, float]:
        """
        Check if a set of layers is compatible for coexistence.

        Args:
            layer_set: Set of layers to check

        Returns:
            (is_compatible, compatibility_score)
            where compatibility_score in [0, 1]
        """
        if len(layer_set) <= 1:
            return True, 1.0

        # Compute minimum pairwise compatibility
        min_compatibility = 1.0
        layer_list = list(layer_set)

        for i, layer1 in enumerate(layer_list):
            for layer2 in layer_list[i+1:]:
                compat = self.compatibility_matrix.get((layer1, layer2), 0.8)
                min_compatibility = min(min_compatibility, compat)

        # Compatible if minimum exceeds threshold
        is_compatible = min_compatibility >= 0.5
        return is_compatible, min_compatibility

    def export_model(self) -> dict:
        """Export model to JSON-serializable format."""
        return {
            'influences': [
                {
                    'source': inf.source_layer.value,
                    'target': inf.target_layer.value,
                    'type': inf.influence_type.value,
                    'strength': inf.strength,
                    'formula': inf.functional_form,
                    'conditions': inf.conditions
                }
                for inf in self.influences
            ],
            'compatibility_matrix': {
                f"{l1.value},{l2.value}": score
                for (l1, l2), score in self.compatibility_matrix.items()
            }
        }


def initialize_canonical_activation_rules() -> dict[Layer, LayerActivationRule]:
    """
    Initialize canonical activation rules for all layers.

    These rules encode when each layer becomes relevant based on
    method characteristics.

    Returns:
        Dict mapping Layer to LayerActivationRule
    """
    rules = {}

    # @q activation: Method operates on individual questions
    rules[Layer.QUESTION] = LayerActivationRule(
        layer=Layer.QUESTION,
        triggers=[
```

```
            lambda mc: mc.get('operates_on_questions', False),
            lambda mc: mc.get('question_count', 0) > 0,
        ],
        priority=100  # Highest priority - foundational layer
    )

    # @d activation: Method aggregates across dimensions
    rules[Layer.DIMENSION] = LayerActivationRule(
        layer=Layer.DIMENSION,
        triggers=[
            lambda mc: mc.get('aggregates_dimensions', False),
            lambda mc: mc.get('dimension_count', 0) > 0,
        ],
        prerequisites={Layer.QUESTION},  # Requires question layer
        priority=90
    )

    # @p activation: Method addresses policy areas
    rules[Layer.POLICY_AREA] = LayerActivationRule(
        layer=Layer.POLICY_AREA,
        triggers=[
            lambda mc: mc.get('addresses_policy_areas', False),
            lambda mc: mc.get('policy_area_count', 0) > 0,
            lambda mc: mc.get('dimension_count', 0) >= 3,  # @d → @p influence
        ],
        prerequisites={Layer.DIMENSION},  # Requires dimension layer
        priority=80
    )

    # @C activation: Method uses ensemble techniques
    rules[Layer.CONGRUENCE] = LayerActivationRule(
        layer=Layer.CONGRUENCE,
        triggers=[
            lambda mc: mc.get('uses_ensemble', False),
            lambda mc: mc.get('ensemble_method_count', 0) > 1,
        ],
        priority=70
    )

    # @m activation: Method performs cross-layer meta-aggregation
    rules[Layer.META] = LayerActivationRule(
        layer=Layer.META,
        triggers=[
            lambda mc: mc.get('performs_meta_aggregation', False),
            lambda mc: len(mc.get('active_base_layers', set())) >= 2,
        ],
        priority=60  # Lowest priority - synthesizes other layers
    )

    return rules


# Global canonical model instance
CANONICAL_LAYER_MODEL = LayerCoexistenceModel()

# Register activation rules
for _layer, rule in initialize_canonical_activation_rules().items():
    CANONICAL_LAYER_MODEL.add_activation_rule(rule)

===== FILE: src/saaaaaa/core/observability/__init__.py =====
"""Observability module for F.A.R.F.A.N runtime monitoring."""

from saaaaaa.core.observability.structured_logging import log_fallback, get_logger
from saaaaaa.core.observability.metrics import (
    increment_fallback,
    increment_segmentation_method,
    increment_calibration_mode,
    increment_document_id_source,
```

```
    increment_hash_algo,
    increment_graph_metrics_skipped,
    increment_contradiction_mode,
)

__all__ = [
    "log_fallback",
    "get_logger",
    "increment_fallback",
    "increment_segmentation_method",
    "increment_calibration_mode",
    "increment_document_id_source",
    "increment_hash_algo",
    "increment_graph_metrics_skipped",
    "increment_contradiction_mode",
]

===== FILE: src/saaaaaa/core/observability/metrics.py =====
"""
Prometheus-style metrics for F.A.R.F.A.N runtime observability.

This module provides metric counters for tracking fallbacks, degradations,
and runtime behavior. All metrics are exposed via the ATROZ dashboard's
hidden layer for centralized health monitoring.

Metrics are prefixed with 'saaaaaa_' and include labels for categorization.
"""

from typing import Optional
from prometheus_client import Counter, Gauge, Histogram

from saaaaaa.core.runtime_config import RuntimeMode
from saaaaaa.core.contracts.runtime_contracts import (
    FallbackCategory,
    SegmentationMethod,
    CalibrationMode,
    DocumentIdSource,
)


# ============================================================================
# Fallback Metrics
# ============================================================================

fallback_activations_total = Counter(
    'saaaaaa_fallback_activations_total',
    'Total number of fallback activations',
    ['component', 'fallback_category', 'fallback_mode', 'runtime_mode']
)


def increment_fallback(
    component: str,
    fallback_category: FallbackCategory,
    fallback_mode: str,
    runtime_mode: RuntimeMode,
) -> None:
    """
    Increment fallback activation counter.

    Args:
        component: Component name
        fallback_category: Fallback category (A/B/C/D)
        fallback_mode: Specific fallback mode
        runtime_mode: Current runtime mode
    """
    fallback_activations_total.labels(
        component=component,
```

```
        fallback_category=fallback_category.value,
        fallback_mode=fallback_mode,
        runtime_mode=runtime_mode.value,
    ).inc()


# ============================================================================
# Segmentation Metrics
# ============================================================================

segmentation_method_total = Counter(
    'saaaaaa_segmentation_method_total',
    'Total segmentation method usage',
    ['method', 'runtime_mode']
)


def increment_segmentation_method(
    method: SegmentationMethod,
    runtime_mode: RuntimeMode,
) -> None:
    """
    Increment segmentation method counter.

    Args:
        method: Segmentation method used
        runtime_mode: Current runtime mode
    """
    segmentation_method_total.labels(
        method=method.value,
        runtime_mode=runtime_mode.value,
    ).inc()


# ============================================================================
# Calibration Metrics
# ============================================================================

calibration_mode_total = Counter(
    'saaaaaa_calibration_mode_total',
    'Total calibration mode usage',
    ['mode', 'runtime_mode']
)


def increment_calibration_mode(
    mode: CalibrationMode,
    runtime_mode: RuntimeMode,
) -> None:
    """
    Increment calibration mode counter.

    Args:
        mode: Calibration mode
        runtime_mode: Current runtime mode
    """
    calibration_mode_total.labels(
        mode=mode.value,
        runtime_mode=runtime_mode.value,
    ).inc()


# ============================================================================
# Document ID Metrics
# ============================================================================

document_id_source_total = Counter(
    'saaaaaa_document_id_source_total',
```

```
    'Total document ID source usage',
    ['source', 'runtime_mode']
)


def increment_document_id_source(
    source: DocumentIdSource,
    runtime_mode: RuntimeMode,
) -> None:
    """
    Increment document ID source counter.

    Args:
        source: Document ID source
        runtime_mode: Current runtime mode
    """
    document_id_source_total.labels(
        source=source.value,
        runtime_mode=runtime_mode.value,
    ).inc()


# ============================================================================
# Hash Algorithm Metrics
# ============================================================================

hash_algo_total = Counter(
    'saaaaaa_hash_algo_total',
    'Total hash algorithm usage',
    ['algo', 'runtime_mode']
)


def increment_hash_algo(
    algo: str,
    runtime_mode: RuntimeMode,
) -> None:
    """
    Increment hash algorithm counter.

    Args:
        algo: Hash algorithm name (e.g., "blake3", "sha256")
        runtime_mode: Current runtime mode
    """
    hash_algo_total.labels(
        algo=algo,
        runtime_mode=runtime_mode.value,
    ).inc()


# ============================================================================
# Graph Metrics
# ============================================================================

graph_metrics_skipped_total = Counter(
    'saaaaaa_graph_metrics_skipped_total',
    'Total graph metrics computation skips',
    ['runtime_mode', 'reason']
)


def increment_graph_metrics_skipped(
    reason: str,
    runtime_mode: RuntimeMode,
) -> None:
    """
    Increment graph metrics skipped counter.
```

```
    Args:
        reason: Reason for skip (e.g., "networkx_unavailable")
        runtime_mode: Current runtime mode
    """
    graph_metrics_skipped_total.labels(
        runtime_mode=runtime_mode.value,
        reason=reason,
    ).inc()


# ============================================================================
# Contradiction Detection Metrics
# ============================================================================

contradiction_mode_total = Counter(
    'saaaaaa_contradiction_mode_total',
    'Total contradiction detection mode usage',
    ['mode', 'runtime_mode']
)


def increment_contradiction_mode(
    mode: str,
    runtime_mode: RuntimeMode,
) -> None:
    """
    Increment contradiction mode counter.

    Args:
        mode: Contradiction mode ("full" or "fallback")
        runtime_mode: Current runtime mode
    """
    contradiction_mode_total.labels(
        mode=mode,
        runtime_mode=runtime_mode.value,
    ).inc()


# ============================================================================
# Boot Check Metrics
# ============================================================================

boot_check_failures_total = Counter(
    'saaaaaa_boot_check_failures_total',
    'Total boot check failures',
    ['check_name', 'runtime_mode', 'code']
)


def increment_boot_check_failure(
    check_name: str,
    code: str,
    runtime_mode: RuntimeMode,
) -> None:
    """
    Increment boot check failure counter.

    Args:
        check_name: Name of failed check
        code: Error code
        runtime_mode: Current runtime mode
    """
    boot_check_failures_total.labels(
        check_name=check_name,
        runtime_mode=runtime_mode.value,
        code=code,
    ).inc()
```

```
# ============================================================================
# Pipeline Execution Metrics
# ============================================================================

pipeline_executions_total = Counter(
    'saaaaaa_pipeline_executions_total',
    'Total pipeline executions',
    ['runtime_mode', 'status']
)

pipeline_execution_duration_seconds = Histogram(
    'saaaaaa_pipeline_execution_duration_seconds',
    'Pipeline execution duration in seconds',
    ['runtime_mode'],
    buckets=(1, 5, 10, 30, 60, 120, 300, 600, 1800, 3600)
)


def increment_pipeline_execution(
    runtime_mode: RuntimeMode,
    status: str,
) -> None:
    """
    Increment pipeline execution counter.

    Args:
        runtime_mode: Current runtime mode
        status: Execution status ("success", "failure", "aborted")
    """
    pipeline_executions_total.labels(
        runtime_mode=runtime_mode.value,
        status=status,
    ).inc()


def observe_pipeline_duration(
    duration_seconds: float,
    runtime_mode: RuntimeMode,
) -> None:
    """
    Observe pipeline execution duration.

    Args:
        duration_seconds: Execution duration in seconds
        runtime_mode: Current runtime mode
    """
    pipeline_execution_duration_seconds.labels(
        runtime_mode=runtime_mode.value,
    ).observe(duration_seconds)


# ============================================================================
# ATROZ Dashboard Integration
# ============================================================================

# Current runtime mode gauge (for dashboard display)
current_runtime_mode = Gauge(
    'saaaaaa_current_runtime_mode',
    'Current runtime mode (0=PROD, 1=DEV, 2=EXPLORATORY)',
    []
)


def set_current_runtime_mode(mode: RuntimeMode) -> None:
    """
    Set current runtime mode gauge for dashboard display.
```

```
    Args:
        mode: Current runtime mode
    """
    mode_value = {
        RuntimeMode.PROD: 0,
        RuntimeMode.DEV: 1,
        RuntimeMode.EXPLORATORY: 2,
    }[mode]
    current_runtime_mode.set(mode_value)


# Health status gauge (for dashboard alerts)
system_health_status = Gauge(
    'saaaaaa_system_health_status',
    'System health status (0=degraded, 1=healthy)',
    []
)


def set_system_health_status(healthy: bool) -> None:
    """
    Set system health status for dashboard alerts.

    Args:
        healthy: Whether system is healthy (no Category A fallbacks)
    """
    system_health_status.set(1 if healthy else 0)


# ============================================================================
# Metric Export for ATROZ Dashboard
# ============================================================================

def get_all_metrics() -> dict[str, any]:
    """
    Get all metrics for ATROZ dashboard integration.

    Returns:
        Dictionary of all metric collectors
    """
    return {
        'fallback_activations_total': fallback_activations_total,
        'segmentation_method_total': segmentation_method_total,
        'calibration_mode_total': calibration_mode_total,
        'document_id_source_total': document_id_source_total,
        'hash_algo_total': hash_algo_total,
        'graph_metrics_skipped_total': graph_metrics_skipped_total,
        'contradiction_mode_total': contradiction_mode_total,
        'boot_check_failures_total': boot_check_failures_total,
        'pipeline_executions_total': pipeline_executions_total,
        'pipeline_execution_duration_seconds': pipeline_execution_duration_seconds,
        'current_runtime_mode': current_runtime_mode,
        'system_health_status': system_health_status,
    }
```

```
===== FILE: src/saaaaaa/core/observability/structured_logging.py =====
"""
Structured logging for F.A.R.F.A.N runtime observability.

This module provides standardized logging for fallbacks, degradations, and
runtime events with consistent field injection for observability.
"""

import logging
import structlog
from typing import Optional

from saaaaaa.core.runtime_config import RuntimeMode
```

```
from saaaaaa.core.contracts.runtime_contracts import FallbackCategory


# Configure structlog for JSON logging
structlog.configure(
    processors=[
        structlog.stdlib.filter_by_level,
        structlog.stdlib.add_logger_name,
        structlog.stdlib.add_log_level,
        structlog.stdlib.PositionalArgumentsFormatter(),
        structlog.processors.TimeStamper(fmt="iso"),
        structlog.processors.StackInfoRenderer(),
        structlog.processors.format_exc_info,
        structlog.processors.UnicodeDecoder(),
        structlog.processors.JSONRenderer()
    ],
    context_class=dict,
    logger_factory=structlog.stdlib.LoggerFactory(),
    cache_logger_on_first_use=True,
)


def get_logger(name: str = __name__) -> structlog.BoundLogger:
    """
    Get a structured logger instance.

    Args:
        name: Logger name (typically __name__)

    Returns:
        Structured logger instance
    """
    return structlog.get_logger(name)


def log_fallback(
    component: str,
    subsystem: str,
    fallback_category: FallbackCategory,
    fallback_mode: str,
    reason: str,
    runtime_mode: RuntimeMode,
    document_id: Optional[str] = None,
    run_id: Optional[str] = None,
    logger: Optional[structlog.BoundLogger] = None,
) -> None:
    """
    Emit structured log for fallback activation.

    This is the standard logging function for all fallback events. It ensures
    consistent field injection and proper categorization for observability.

    Args:
        component: Component name (e.g., "language_detection", "calibration")
        subsystem: Subsystem name (e.g., "spc_ingestion", "orchestrator")
        fallback_category: Fallback category (A/B/C/D)
        fallback_mode: Specific fallback mode (e.g., "warn_default_es", "estimated")
        reason: Human-readable reason for fallback
        runtime_mode: Current runtime mode
        document_id: Optional document identifier
        run_id: Optional run identifier
        logger: Optional logger instance (creates new if None)

    Example:
        >>> log_fallback(
        ...     component="language_detection",
        ...     subsystem="spc_ingestion",
        ...     fallback_category=FallbackCategory.B,
```

```
        ...     fallback_mode="warn_default_es",
        ...     reason="LangDetectException",
        ...     runtime_mode=RuntimeMode.PROD
        ... )
    """
    if logger is None:
        logger = get_logger()

    # Determine log level based on category
    if fallback_category == FallbackCategory.A:
        log_level = logging.ERROR
    elif fallback_category == FallbackCategory.B:
        log_level = logging.WARNING
    else:
        log_level = logging.INFO

    # Emit structured log
    logger.log(
        log_level,
        "fallback_activated",
        component=component,
        subsystem=subsystem,
        fallback_category=fallback_category.value,
        fallback_mode=fallback_mode,
        reason=reason,
        runtime_mode=runtime_mode.value,
        document_id=document_id,
        run_id=run_id,
    )


def log_boot_check_failure(
    check_name: str,
    reason: str,
    code: str,
    runtime_mode: RuntimeMode,
    logger: Optional[structlog.BoundLogger] = None,
) -> None:
    """
    Log boot check failure.

    Args:
        check_name: Name of failed check
        reason: Failure reason
        code: Error code
        runtime_mode: Current runtime mode
        logger: Optional logger instance
    """
    if logger is None:
        logger = get_logger()

    logger.error(
        "boot_check_failed",
        check_name=check_name,
        reason=reason,
        code=code,
        runtime_mode=runtime_mode.value,
    )


def log_boot_check_success(
    check_name: str,
    runtime_mode: RuntimeMode,
    logger: Optional[structlog.BoundLogger] = None,
) -> None:
    """
    Log boot check success.
```

```
    Args:
        check_name: Name of successful check
        runtime_mode: Current runtime mode
        logger: Optional logger instance
    """
    if logger is None:
        logger = get_logger()

    logger.info(
        "boot_check_passed",
        check_name=check_name,
        runtime_mode=runtime_mode.value,
    )


def log_runtime_config_loaded(
    config_repr: str,
    runtime_mode: RuntimeMode,
    logger: Optional[structlog.BoundLogger] = None,
) -> None:
    """
    Log runtime configuration loaded.

    Args:
        config_repr: String representation of config
        runtime_mode: Runtime mode
        logger: Optional logger instance
    """
    if logger is None:
        logger = get_logger()

    logger.info(
        "runtime_config_loaded",
        config=config_repr,
        runtime_mode=runtime_mode.value,
    )

===== FILE: src/saaaaaa/core/orchestrator/__init__.py =====
"""Orchestrator utilities with contract validation on import."""
from __future__ import annotations

from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from .questionnaire import CanonicalQuestionnaire

# Import core classes from the refactored package
from .core import (
    AbortRequested,
    AbortSignal,
    Evidence,
    MethodExecutor,
    MicroQuestionRun,
    Orchestrator,
    PhaseInstrumentation,
    PhaseResult,
    PreprocessedDocument,
    ResourceLimits,
    ScoredMicroQuestion,
)
from .evidence_registry import (
    EvidenceRecord,
    EvidenceRegistry,
    ProvenanceDAG,
    ProvenanceNode,
    get_global_registry,
)
```

```
__all__ = [
    "EvidenceRecord",
    "EvidenceRegistry",
    "ProvenanceDAG",
    "ProvenanceNode",
    "get_global_registry",
    "Orchestrator",
    "MethodExecutor",
    "PreprocessedDocument",
    "Evidence",
    "AbortSignal",
    "AbortRequested",
    "ResourceLimits",
    "PhaseInstrumentation",
    "PhaseResult",
    "MicroQuestionRun",
    "ScoredMicroQuestion",
]

===== FILE: src/saaaaaa/core/orchestrator/arg_router.py =====
"""Argument routing with special routes, strict validation, and comprehensive metrics.

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

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

from __future__ import annotations

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

import structlog

logger = structlog.get_logger(__name__)
std_logger = logging.getLogger(__name__)

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


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

class ArgRouterError(RuntimeError):
```

```
    """Base exception for routing and validation issues."""


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

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


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

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


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

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

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

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

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

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

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

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

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

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

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

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

        remaining = dict(payload)

        for param in spec.positional:
            if param.name not in remaining:
                if param.required:
                    missing = {param.name}
                    raise ArgumentValidationError(
                        class_name,
                        method_name,
                        missing=missing,
                    )
                continue
```

```
            value = remaining.pop(param.name)
            if not self._matches_annotation(value, param.annotation):
                expected = self._describe_annotation(param.annotation)
                type_mismatches[param.name] = expected
            args.append(value)

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

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

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

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

        return tuple(args), kwargs

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

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

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

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

```
    has_var_positional = False

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

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

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

```
            if not isinstance(value, set):
                return False
            if not args:
                return True
            return all(ArgRouter._matches_annotation(item, args[0]) for item in value)
        if origin in (dict, dict):
            if not isinstance(value, dict):
                return False
            if len(args) != 2:
                return True
            key_type, value_type = args
            return all(
                ArgRouter._matches_annotation(k, key_type)
                and ArgRouter._matches_annotation(v, value_type)
                for k, v in value.items()
            )
        if origin is Union:
            return any(ArgRouter._matches_annotation(value, arg) for arg in args)
        return True

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


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

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

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

    @classmethod
    def from_env(cls) -> PayloadDriftMonitor:
        enabled = os.getenv("ORCHESTRATOR_SAMPLING_VALIDATION", "").lower() in {
            "1",
            "true",
```

```
            "yes",
            "on",
        }
        try:
            sample_rate = float(os.getenv("ORCHESTRATOR_SAMPLING_RATE", "0.05"))
        except ValueError:
            sample_rate = 0.05
        return cls(sample_rate=sample_rate, enabled=enabled)

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

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

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


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


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

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


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

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

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

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

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

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

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

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

```
Returns:
    Dict mapping method names to route specs
"""
routes = {
    "_extract_quantitative_claims": {
        "required_args": ["content"],
        "optional_args": ["context", "thresholds", "patterns"],
        "accepts_kwargs": True,
        "description": "Extract quantitative claims from content",
    },
    "_parse_number": {
        "required_args": ["text"],
        "optional_args": ["locale", "unit_system"],
        "accepts_kwargs": True,
        "description": "Parse numerical value from text",
    },
    "_determine_semantic_role": {
        "required_args": ["text", "context"],
        "optional_args": ["role_taxonomy", "confidence_threshold"],
        "accepts_kwargs": True,
        "description": "Determine semantic role of text element",
    },
    "_compile_pattern_registry": {
        "required_args": ["patterns"],
        "optional_args": ["category", "flags"],
        "accepts_kwargs": False,
        "description": "Compile patterns into regex registry",
    },
    "_analyze_temporal_coherence": {
        "required_args": ["content"],
        "optional_args": ["temporal_patterns", "baseline_date"],
        "accepts_kwargs": True,
        "description": "Analyze temporal coherence of content",
    },
    "_validate_evidence_chain": {
        "required_args": ["claims", "evidence"],
        "optional_args": ["validation_rules", "min_confidence"],
        "accepts_kwargs": True,
        "description": "Validate evidence chain for claims",
    },
    "_calculate_confidence_score": {
        "required_args": ["evidence"],
        "optional_args": ["prior", "weights"],
        "accepts_kwargs": True,
        "description": "Calculate Bayesian confidence score",
    },
    "_extract_indicators": {
        "required_args": ["content"],
        "optional_args": ["indicator_patterns", "extraction_mode"],
        "accepts_kwargs": True,
        "description": "Extract KPI indicators from content",
    },
    "_parse_temporal_reference": {
        "required_args": ["text"],
        "optional_args": ["reference_date", "format_hints"],
        "accepts_kwargs": True,
        "description": "Parse temporal reference from text",
    },
    "_determine_policy_area": {
        "required_args": ["content"],
        "optional_args": ["taxonomy", "multi_label"],
        "accepts_kwargs": True,
        "description": "Classify content into policy area",
    },
    "_compile_regex_patterns": {
        "required_args": ["pattern_list"],
        "optional_args": ["flags", "validate"],
        "accepts_kwargs": False,
```

```
    "description": "Compile list of regex patterns",
},
"_analyze_source_reliability": {
    "required_args": ["source"],
    "optional_args": ["source_patterns", "reliability_threshold"],
    "accepts_kwargs": True,
    "description": "Analyze reliability of information source",
```