```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 11
================================================================================
Generated: 2025-12-07T06:17:18.917142
Files in this batch: 17
================================================================================


================================================================================
FILE: src/farfan_pipeline/contracts/verify_all_contracts.py
================================================================================

#!/usr/bin/env python3
"""
Verification Script for 15-Contract Suite
Runs all tests and tools, then validates certificates.
"""
import glob
import json
import os
import subprocess
import sys

CONTRACTS_DIR = "src/farfan_pipeline/contracts"
TOOLS_DIR = os.path.join(CONTRACTS_DIR, "tools")
TESTS_DIR = os.path.join(CONTRACTS_DIR, "tests")


def run_command(cmd: str, description: str, set_pythonpath: bool = False) -> bool:
    print(f"Running {description}...")
    try:
        env = os.environ.copy()
        if set_pythonpath:
            cwd = os.getcwd()
            src_path = os.path.join(cwd, "src")
            env["PYTHONPATH"] = f"{src_path}:{env.get('PYTHONPATH', '')}"
        subprocess.check_call(cmd, shell=True, env=env)
        print(f"â\234\205 {description} PASSED")
        return True
    except subprocess.CalledProcessError:
        print(f"â\235\214 {description} FAILED")
        return False


def main() -> None:
    print("=== STARTING VERIFICATION OF 15-CONTRACT SUITE ===")

    # 1. Run Pytest Suite
    print("\n--- 1. RUNNING TESTS ---")
    if not run_command(
        f"pytest {TESTS_DIR} -v", "All Contract Tests", set_pythonpath=True
    ):
        sys.exit(1)

    # 2. Run CLI Tools to generate certificates
    print("\n--- 2. GENERATING CERTIFICATES ---")
```

```
    tools = glob.glob(os.path.join(TOOLS_DIR, "*.py"))
    for tool in sorted(tools):
        tool_name = os.path.basename(tool)
        if not run_command(f"python {tool}", f"Tool: {tool_name}", set_pythonpath=True):
            sys.exit(1)

    # 3. Verify Certificates
    print("\n--- 3. VERIFYING CERTIFICATES ---")
    expected_certs = [
        "rc_certificate.json",
        "sc_certificate.json",
        "cic_certificate.json",
        "pic_certificate.json",
        "bmc_certificate.json",
        "toc_certificate.json",
        "rec_certificate.json",
        "asc_certificate.json",
        "idc_certificate.json",
        "rcc_certificate.json",
        "mcc_certificate.json",
        "ffc_certificate.json",
        "cdc_certificate.json",
        "tc_certificate.json",
        "refc_certificate.json",
    ]

    all_passed = True
    for cert_file in expected_certs:
        if not os.path.exists(cert_file):
            print(f"â\235\214 {cert_file}: MISSING")
            all_passed = False
            continue

        try:
            with open(cert_file) as f:
                data = json.load(f)
                if data.get("pass") is True:
                    print(f"â\234\205 {cert_file}: PASS")
                else:
                    print(f"â\235\214 {cert_file}: FAIL (pass != true)")
                    all_passed = False
        except Exception as e:
            print(f"â\235\214 {cert_file}: ERROR ({e})")
            all_passed = False

    if all_passed:
        print("\n=== ALL SYSTEM CONTRACTS VERIFIED SUCCESSFULLY ===")
        sys.exit(0)
    else:
        print("\n=== VERIFICATION FAILED ===")
        sys.exit(1)


if __name__ == "__main__":
    main()

```

```
113:
114:
115: ================================================================================
116: FILE: src/farfan_pipeline/contracts.py
117: ================================================================================
118:
119: """Contracts module re-exports from utils.contracts package.
120:
121: This module provides backward compatibility for code that imports
122: contracts from farfan_core.contracts instead of farfan_core.utils.contracts.
123: """
124:
125: from farfan_pipeline.utils.contracts import (
126:     MISSING,
127:     AnalysisInputV1,
128:     AnalysisInputV1Optional,
129:     AnalysisOutputV1,
130:     AnalysisOutputV1Optional,
131:     AnalyzerProtocol,
132:     ContractMismatchError,
133:     DocumentLoaderProtocol,
134:     DocumentMetadataV1,
135:     DocumentMetadataV1Optional,
136:     ExecutionContextV1,
137:     ExecutionContextV1Optional,
138:     ProcessedTextV1,
139:     ProcessedTextV1Optional,
140:     SentenceCollection,
141:     TextDocument,
142:     TextProcessorProtocol,
143:     ensure_hashable,
144:     ensure_iterable_not_string,
145:     validate_contract,
146:     validate_mapping_keys,
147: )
148:
149: __all__ = [
150:     "MISSING",
151:     "AnalysisInputV1",
152:     "AnalysisInputV1Optional",
153:     "AnalysisOutputV1",
154:     "AnalysisOutputV1Optional",
155:     "AnalyzerProtocol",
156:     "ContractMismatchError",
157:     "DocumentLoaderProtocol",
158:     "DocumentMetadataV1",
159:     "DocumentMetadataV1Optional",
160:     "ExecutionContextV1",
161:     "ExecutionContextV1Optional",
162:     "ProcessedTextV1",
163:     "ProcessedTextV1Optional",
164:     "SentenceCollection",
165:     "TextDocument",
166:     "TextProcessorProtocol",
167:     "ensure_hashable",
168:     "ensure_iterable_not_string",
```

```
    "validate_contract",
    "validate_mapping_keys",
]


================================================================================
FILE: src/farfan_pipeline/controls/__init__.py
================================================================================

"""Controls module for system validation and checks."""


================================================================================
FILE: src/farfan_pipeline/controls/graphs/__init__.py
================================================================================

"""Graph-based controls and analysis."""


================================================================================
FILE: src/farfan_pipeline/core/__init__.py
================================================================================

"""Core components of the SAAAAAA system."""


================================================================================
FILE: src/farfan_pipeline/core/aggregation.py
================================================================================

"""Aggregation module re-exports from processing package.

This module provides backward compatibility for code that imports
aggregation classes from farfan_core.core.aggregation instead of
farfan_core.processing.aggregation.
"""

from farfan_pipeline.processing.aggregation import (
    AreaPolicyAggregator,
    AreaScore,
    ClusterAggregator,
    ClusterScore,
    DimensionAggregator,
    DimensionScore,
    MacroAggregator,
    ScoredResult,
)

__all__ = [
    "AreaPolicyAggregator",
    "AreaScore",
    "ClusterAggregator",
```

```
    "ClusterScore",
    "DimensionAggregator",
    "DimensionScore",
    "MacroAggregator",
    "ScoredResult",
]



================================================================================
FILE: src/farfan_pipeline/core/analysis_port.py
================================================================================

"""
Port interface for recommendation engine.

Defines the abstract interface for recommendation generation, following
the Ports and Adapters (Hexagonal) architecture pattern.

Version: 1.0.0
"""

from typing import Any, Protocol


class RecommendationEnginePort(Protocol):
    """Port for recommendation engine operations.

    Provides abstract interface for multi-level recommendation generation.
    Core orchestrator depends on this port, not the concrete implementation.

    Implementations must support three levels of recommendations:
    - MICRO: Question-level recommendations (PA-DIM combinations)
    - MESO: Cluster-level recommendations (CL01-CL04)
    - MACRO: Plan-level strategic recommendations
    """

    def generate_all_recommendations(
        self,
        micro_scores: dict[str, float],
        cluster_data: dict[str, Any],
        macro_data: dict[str, Any],
        context: dict[str, Any] | None = None,
    ) -> dict[str, Any]:
        """Generate recommendations at all three levels.

        Args:
            micro_scores: Dictionary mapping "PA##-DIM##" to scores (0.0-3.0)
            cluster_data: Dictionary with cluster metrics:
                {
                    'CL01': {'score': 75.0, 'variance': 0.15, 'weak_pa': 'PA02'},
                    'CL02': {'score': 62.0, 'variance': 0.22, 'weak_pa': 'PA05'},
                    ...
                }
            macro_data: Dictionary with macro-level metrics:
                {
```

```
                    'macro_score': 68.5,
                    'cross_cutting_coherence': 0.72,
                    'systemic_gaps': ['gap1', 'gap2'],
                    'strategic_alignment': 0.65
                }
            context: Optional context for template rendering

        Returns:
            Dictionary mapping level to RecommendationSet:
            {
                'MICRO': RecommendationSet(...),
                'MESO': RecommendationSet(...),
                'MACRO': RecommendationSet(...)
            }

        Requires:
            - micro_scores keys follow "PA##-DIM##" format
            - cluster_data keys follow "CL##" format
            - macro_data contains required metrics

        Ensures:
            - Returns dict with 'MICRO', 'MESO', 'MACRO' keys
            - Each value is a RecommendationSet with to_dict() method
            - Generated_at timestamp is present
        """
        ...

    def generate_micro_recommendations(
        self, scores: dict[str, float], context: dict[str, Any] | None = None
    ) -> Any:
        """Generate MICRO-level recommendations.

        Args:
            scores: Dictionary mapping "PA##-DIM##" to scores (0.0-3.0)
            context: Optional context for template rendering

        Returns:
            RecommendationSet with MICRO recommendations

        Requires:
            - scores keys follow "PA##-DIM##" format

        Ensures:
            - Returns RecommendationSet with level='MICRO'
            - Contains list of matched recommendations
        """
        ...

    def generate_meso_recommendations(
        self, cluster_data: dict[str, Any], context: dict[str, Any] | None = None
    ) -> Any:
        """Generate MESO-level recommendations.

        Args:
            cluster_data: Dictionary with cluster metrics
            context: Optional context for template rendering
```

```
337:
338:         Returns:
339:             RecommendationSet with MESO recommendations
340:
341:         Requires:
342:             - cluster_data keys follow "CL##" format
343:             - Each cluster has score, variance, weak_pa fields
344:
345:         Ensures:
346:             - Returns RecommendationSet with level='MESO'
347:             - Contains list of matched recommendations
348:         """
349:         ...
350:
351:     def generate_macro_recommendations(
352:         self, macro_data: dict[str, Any], context: dict[str, Any] | None = None
353:     ) -> Any:
354:         """Generate MACRO-level recommendations.
355:
356:         Args:
357:             macro_data: Dictionary with macro-level metrics
358:             context: Optional context for template rendering
359:
360:         Returns:
361:             RecommendationSet with MACRO recommendations
362:
363:         Requires:
364:             - macro_data contains macro_score
365:             - macro_data contains strategic metrics
366:
367:         Ensures:
368:             - Returns RecommendationSet with level='MACRO'
369:             - Contains list of matched recommendations
370:         """
371:         ...
372:
373:     def reload_rules(self) -> None:
374:         """Reload recommendation rules from disk.
375:
376:         Enables hot-reloading of rules during development/operation.
377:
378:         Ensures:
379:             - Rules are revalidated against schema
380:             - Engine state is consistent after reload
381:         """
382:         ...
383:
384:
385: __all__ = ["RecommendationEnginePort"]
386:
387:
388:
389: ================================================================================
390: FILE: src/farfan_pipeline/core/boot_checks.py
391: ================================================================================
392:
```

```
"""
Boot-time validation checks for F.A.R.F.A.N runtime dependencies.

This module provides boot checks that validate critical dependencies before
pipeline execution. In PROD mode, failures abort execution unless explicitly
allowed by configuration flags.
"""

import importlib
import json
from pathlib import Path
from typing import Optional

from farfan_pipeline.core.runtime_config import RuntimeConfig, RuntimeMode


class BootCheckError(Exception):
    """
    Raised when a boot check fails in strict mode.

    Attributes:
        component: Component that failed (e.g., "contradiction_module")
        reason: Human-readable failure reason
        code: Machine-readable error code
    """

    def __init__(self, component: str, reason: str, code: str):
        self.component = component
        self.reason = reason
        self.code = code
        super().__init__(f"Boot check failed [{code}] {component}: {reason}")


def check_contradiction_module_available(config: RuntimeConfig) -> bool:
    """
    Check if contradiction detection module is available.

    Args:
        config: Runtime configuration

    Returns:
        True if module available or fallback allowed

    Raises:
        BootCheckError: If module unavailable in strict PROD mode
    """
    try:
        from farfan_pipeline.analysis.contradiction_detection import PolicyContradictionDetector
        return True
    except ImportError as e:
        if config.mode == RuntimeMode.PROD and not config.allow_contradiction_fallback:
            raise BootCheckError(
                component="contradiction_module",
                reason=f"PolicyContradictionDetector not available: {e}",
                code="CONTRADICTION_MODULE_MISSING"
            )
```

```
449:         # Fallback allowed in DEV/EXPLORATORY or with flag
450:         return False
451:
452:
453: def check_wiring_validator_available(config: RuntimeConfig) -> bool:
454:     """
455:     Check if WiringValidator is available.
456:
457:     Args:
458:         config: Runtime configuration
459:
460:     Returns:
461:         True if validator available or disable allowed
462:
463:     Raises:
464:         BootCheckError: If validator unavailable in strict PROD mode
465:     """
466:     try:
467:         from farfan_pipeline.core.wiring.validator import WiringValidator
468:         return True
469:     except ImportError as e:
470:         if config.mode == RuntimeMode.PROD and not config.allow_validator_disable:
471:             raise BootCheckError(
472:                 component="wiring_validator",
473:                 reason=f"WiringValidator not available: {e}",
474:                 code="WIRING_VALIDATOR_MISSING"
475:             )
476:         return False
477:
478:
479: def check_spacy_model_available(model_name: str, config: RuntimeConfig) -> bool:
480:     """
481:     Check if preferred spaCy model is installed.
482:
483:     Args:
484:         model_name: spaCy model name (e.g., "es_core_news_lg")
485:         config: Runtime configuration
486:
487:     Returns:
488:         True if model available
489:
490:     Raises:
491:         BootCheckError: If model unavailable in PROD mode
492:     """
493:     try:
494:         import spacy
495:         spacy.load(model_name)
496:         return True
497:     except (ImportError, OSError) as e:
498:         if config.mode == RuntimeMode.PROD:
499:             raise BootCheckError(
500:                 component="spacy_model",
501:                 reason=f"spaCy model '{model_name}' not available: {e}",
502:                 code="SPACY_MODEL_MISSING"
503:             )
504:         return False
```

```
505:
506:
507: def check_calibration_files(config: RuntimeConfig, calibration_dir: Optional[Path] = None) -> bool:
508:     """
509:     Check calibration files exist and have required structure.
510:
511:     Args:
512:         config: Runtime configuration
513:         calibration_dir: Directory containing calibration files (default: config/layer_calibrations)
514:
515:     Returns:
516:         True if calibration files valid
517:
518:     Raises:
519:         BootCheckError: If calibration invalid in strict PROD mode
520:     """
521:     if calibration_dir is None:
522:         calibration_dir = Path("config/layer_calibrations")
523:
524:     if not calibration_dir.exists():
525:         if config.mode == RuntimeMode.PROD and config.strict_calibration:
526:             raise BootCheckError(
527:                 component="calibration_files",
528:                 reason=f"Calibration directory not found: {calibration_dir}",
529:                 code="CALIBRATION_DIR_MISSING"
530:             )
531:         return False
532:
533:     # Check for required calibration files
534:     required_files = ["intrinsic_calibration.json", "fusion_specification.json"]
535:     missing_files = []
536:
537:     for filename in required_files:
538:         file_path = calibration_dir.parent / filename
539:         if not file_path.exists():
540:             missing_files.append(filename)
541:
542:     if missing_files:
543:         if config.mode == RuntimeMode.PROD and config.strict_calibration:
544:             raise BootCheckError(
545:                 component="calibration_files",
546:                 reason=f"Missing required calibration files: {', '.join(missing_files)}",
547:                 code="CALIBRATION_FILES_MISSING"
548:             )
549:         return False
550:
551:     # Validate _base_weights presence in strict mode
552:     if config.strict_calibration:
553:         intrinsic_path = calibration_dir.parent / "intrinsic_calibration.json"
554:         try:
555:             with open(intrinsic_path) as f:
556:                 data = json.load(f)
557:
558:             if "_base_weights" not in data:
559:                 if config.mode == RuntimeMode.PROD:
560:                     raise BootCheckError(
```

```
                        component="calibration_files",
                        reason=f"Missing _base_weights in {intrinsic_path}",
                        code="CALIBRATION_BASE_WEIGHTS_MISSING"
                    )
                return False
        except (json.JSONDecodeError, IOError) as e:
            if config.mode == RuntimeMode.PROD:
                raise BootCheckError(
                    component="calibration_files",
                    reason=f"Failed to parse {intrinsic_path}: {e}",
                    code="CALIBRATION_PARSE_ERROR"
                )
            return False

    return True


def check_orchestration_metrics_contract(config: RuntimeConfig) -> bool:
    """
    Check that orchestration metrics contract is properly defined.

    This validates that the _execution_metrics['phase_2'] schema exists
    and is properly structured.

    Args:
        config: Runtime configuration

    Returns:
        True if contract valid

    Raises:
        BootCheckError: If contract invalid in PROD mode
    """
    try:
        # Import orchestrator to check metrics contract
        from farfan_pipeline.core.orchestrator.core import Orchestrator

        # Verify phase_2 metrics schema exists
        # This is a placeholder - actual implementation would check the schema
        return True
    except ImportError as e:
        if config.mode == RuntimeMode.PROD:
            raise BootCheckError(
                component="orchestration_metrics",
                reason=f"Orchestrator not available: {e}",
                code="ORCHESTRATOR_MISSING"
            )
        return False


def check_networkx_available() -> bool:
    """
    Check if NetworkX is available for graph metrics.

    Returns:
        True if NetworkX available
```

```
617:
618:     Note:
619:         This is a soft check - NetworkX unavailability is Category B (quality degradation)
620:     """
621:     try:
622:         import networkx
623:         return True
624:     except ImportError:
625:         return False
626:
627:
628: def run_boot_checks(config: RuntimeConfig) -> dict[str, bool]:
629:     """
630:     Run all boot checks and return results.
631:
632:     Args:
633:         config: Runtime configuration
634:
635:     Returns:
636:         Dictionary mapping check name to success status
637:
638:     Raises:
639:         BootCheckError: If any critical check fails in strict PROD mode
640:
641:     Example:
642:         >>> config = RuntimeConfig.from_env()
643:         >>> results = run_boot_checks(config)
644:         >>> assert results['contradiction_module']
645:     """
646:     results = {}
647:
648:     # Critical checks (Category A)
649:     results['contradiction_module'] = check_contradiction_module_available(config)
650:     results['wiring_validator'] = check_wiring_validator_available(config)
651:     results['spacy_model'] = check_spacy_model_available(config.preferred_spacy_model, config)
652:     results['calibration_files'] = check_calibration_files(config)
653:     results['orchestration_metrics'] = check_orchestration_metrics_contract(config)
654:
655:     # Quality checks (Category B)
656:     results['networkx'] = check_networkx_available()
657:
658:     return results
659:
660:
661: def get_boot_check_summary(results: dict[str, bool]) -> str:
662:     """
663:     Generate human-readable summary of boot check results.
664:
665:     Args:
666:         results: Boot check results from run_boot_checks()
667:
668:     Returns:
669:         Formatted summary string
670:     """
671:     passed = sum(1 for v in results.values() if v)
672:     total = len(results)
```

```
673:
674:     lines = [f"Boot Checks: {passed}/{total} passed"]
675:     lines.append("")
676:
677:     for check, success in results.items():
678:         status = "â\234\223" if success else "â\234\227"
679:         lines.append(f"  {status} {check}")
680:
681:     return "\n".join(lines)
682:
683:
684:
685: ================================================================================
686: FILE: src/farfan_pipeline/core/calibration/__init__ 2.py
687: ================================================================================
688:
689: """
690: SAAAAAA Calibration System.
691:
692: This package implements the 7-layer method calibration framework:
693: - @b (Base): Intrinsic method quality
694: - @u (Unit): PDT quality
695: - @q, @d, @p (Contextual): Method-context compatibility
696: - @C (Congruence): Method ensemble validation
697: - @chain (Chain): Data flow integrity
698: - @m (Meta): Governance and observability
699:
700: Final scores are produced via Choquet 2-Additive aggregation.
701: """
702:
703: from farfan_pipeline.core.calibration.data_structures import (
704:     LayerID,
705:     LayerScore,
706:     ContextTuple,
707:     CalibrationSubject,
708:     CompatibilityMapping,
709:     InteractionTerm,
710:     CalibrationResult,
711: )
712:
713: from farfan_pipeline.core.calibration.config import (
714:     UnitLayerConfig,
715:     MetaLayerConfig,
716:     ChoquetAggregationConfig,
717:     CalibrationSystemConfig,
718:     DEFAULT_CALIBRATION_CONFIG,
719: )
720:
721: from farfan_pipeline.core.calibration.pdt_structure import PDTStructure
722:
723: from farfan_pipeline.core.calibration.compatibility import (
724:     CompatibilityRegistry,
725:     ContextualLayerEvaluator,
726: )
727:
728: from farfan_pipeline.core.calibration.unit_layer import UnitLayerEvaluator
```

```
729: from farfan_pipeline.core.calibration.congruence_layer import CongruenceLayerEvaluator
730: from farfan_pipeline.core.calibration.chain_layer import ChainLayerEvaluator
731: from farfan_pipeline.core.calibration.meta_layer import MetaLayerEvaluator
732: from farfan_pipeline.core.calibration.choquet_aggregator import ChoquetAggregator
733: from farfan_pipeline.core.calibration.orchestrator import CalibrationOrchestrator
734:
735: __all__ = [
736:     # Data structures
737:     "LayerID",
738:     "LayerScore",
739:     "ContextTuple",
740:     "CalibrationSubject",
741:     "CompatibilityMapping",
742:     "InteractionTerm",
743:     "CalibrationResult",
744:     "PDTStructure",
745:     # Configuration
746:     "UnitLayerConfig",
747:     "MetaLayerConfig",
748:     "ChoquetAggregationConfig",
749:     "CalibrationSystemConfig",
750:     "DEFAULT_CALIBRATION_CONFIG",
751:     # Layer Evaluators
752:     "UnitLayerEvaluator",
753:     "CompatibilityRegistry",
754:     "ContextualLayerEvaluator",
755:     "CongruenceLayerEvaluator",
756:     "ChainLayerEvaluator",
757:     "MetaLayerEvaluator",
758:     # Aggregation & Orchestration
759:     "ChoquetAggregator",
760:     "CalibrationOrchestrator",
761: ]
762:
763:
764:
765: ================================================================================
766: FILE: src/farfan_pipeline/core/calibration/__init__.py
767: ================================================================================
768:
769: """Calibration system - LEGACY STUB for backward compatibility.
770:
771: This module provides minimal stubs to maintain backward compatibility
772: after calibration system cleanup. The actual calibration logic should
773: be migrated to the new structure.
774: """
775:
776: from farfan_pipeline.core.calibration.decorators import calibrated_method
777: from farfan_pipeline.core.calibration.parameter_loader import ParameterLoader, get_parameter_loader
778:
779: __all__ = ["calibrated_method", "ParameterLoader", "get_parameter_loader"]
780:
781:
782:
783: ================================================================================
784: FILE: src/farfan_pipeline/core/calibration/calibration_registry.py
```

```
785: ================================================================================
786:
787: """Calibration Registry Module.
788:
789: This module provides base calibration resolution for orchestrator methods.
790: It defines the MethodCalibration dataclass and functions to resolve calibration
791: parameters for methods, with optional context-aware adjustments.
792:
793: Design Principles:
794: - Base calibration is context-independent
795: - Reads from config/intrinsic_calibration.json
796: - Provides fallback defaults for uncalibrated methods
797: - Supports context-aware resolution via calibration_context module
798: """
799:
800: from __future__ import annotations
801:
802: import json
803: import logging
804: from dataclasses import dataclass
805: from pathlib import Path
806: from typing import Any, Dict, Optional
807:
808: logger = logging.getLogger(__name__)
809:
810: # Canonical repository root
811: # Path hierarchy: calibration_registry.py -> orchestrator -> core -> saaaaaa -> src -> REPO_ROOT
812: _REPO_ROOT = Path(__file__).resolve().parents[4]
813: _CALIBRATION_FILE = _REPO_ROOT / "config" / "intrinsic_calibration.json"
814:
815:
816: @dataclass(frozen=True)
817: class MethodCalibration:
818:     """Calibration parameters for an orchestrator method.
819:
820:     Attributes:
821:         score_min: Minimum score value (typically 0.0)
822:         score_max: Maximum score value (typically 1.0)
823:         min_evidence_snippets: Minimum number of evidence snippets required
824:         max_evidence_snippets: Maximum number of evidence snippets to collect
825:         contradiction_tolerance: Tolerance for contradictory evidence (0.0-1.0)
826:         uncertainty_penalty: Penalty for uncertain evidence (0.0-1.0)
827:         aggregation_weight: Weight in aggregation (typically 1.0)
828:         sensitivity: Method sensitivity to input variations (0.0-1.0)
829:         requires_numeric_support: Whether method requires numeric evidence
830:         requires_temporal_support: Whether method requires temporal evidence
831:         requires_source_provenance: Whether method requires source provenance
832:     """
833:     score_min: float
834:     score_max: float
835:     min_evidence_snippets: int
836:     max_evidence_snippets: int
837:     contradiction_tolerance: float
838:     uncertainty_penalty: float
839:     aggregation_weight: float
840:     sensitivity: float
```

```
    requires_numeric_support: bool
    requires_temporal_support: bool
    requires_source_provenance: bool

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


# Cache for loaded calibration data
_calibration_cache: Optional[Dict[str, Any]] = None


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

    Returns:
        Dictionary containing calibration data
    """
    global _calibration_cache

    if _calibration_cache is not None:
        return _calibration_cache

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

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


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

    Returns:
```

```
897:         Default MethodCalibration with conservative parameters
898:     """
899:     return MethodCalibration(
900:         score_min=0.0,
901:         score_max=1.0,
902:         min_evidence_snippets=3,
903:         max_evidence_snippets=15,
904:         contradiction_tolerance=0.1,
905:         uncertainty_penalty=0.3,
906:         aggregation_weight=1.0,
907:         sensitivity=0.75,
908:         requires_numeric_support=False,
909:         requires_temporal_support=False,
910:         requires_source_provenance=True,
911:     )
912:
913:
914: def resolve_calibration(class_name: str, method_name: str) -> MethodCalibration:
915:     """Resolve base calibration for a method.
916:
917:     This function looks up calibration parameters from the intrinsic calibration
918:     file. If no calibration is found, it returns conservative defaults.
919:
920:     Args:
921:         class_name: Name of the class (e.g., "SemanticAnalyzer")
922:         method_name: Name of the method (e.g., "extract_entities")
923:
924:     Returns:
925:         MethodCalibration with parameters for this method
926:     """
927:     data = _load_calibration_data()
928:
929:     # Try to find calibration for this specific method
930:     method_key = f"{class_name}.{method_name}"
931:
932:     # Check if calibration exists for this method
933:     if method_key in data:
934:         method_data = data[method_key]
935:         try:
936:             return MethodCalibration(
937:                 score_min=method_data.get("score_min", 0.0),
938:                 score_max=method_data.get("score_max", 1.0),
939:                 min_evidence_snippets=method_data.get("min_evidence_snippets", 3),
940:                 max_evidence_snippets=method_data.get("max_evidence_snippets", 15),
941:                 contradiction_tolerance=method_data.get("contradiction_tolerance", 0.1),
942:                 uncertainty_penalty=method_data.get("uncertainty_penalty", 0.3),
943:                 aggregation_weight=method_data.get("aggregation_weight", 1.0),
944:                 sensitivity=method_data.get("sensitivity", 0.75),
945:                 requires_numeric_support=method_data.get("requires_numeric_support", False),
946:                 requires_temporal_support=method_data.get("requires_temporal_support", False),
947:                 requires_source_provenance=method_data.get("requires_source_provenance", True),
948:             )
949:         except (KeyError, ValueError) as e:
950:             logger.warning(f"Invalid calibration for {method_key}: {e}. Using defaults.")
951:             return _get_default_calibration()
952:
```

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


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

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

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

    Returns:
        MethodCalibration with context-aware adjustments applied
    """
    # Get base calibration
    base_calibration = resolve_calibration(class_name, method_name)

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

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

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

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

        # Apply contextual adjustments
        return resolve_contextual_calibration(base_calibration, context)

```

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


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



================================================================================
FILE: src/farfan_pipeline/core/calibration/chain_layer.py
================================================================================

"""
Chain Layer (@chain) - Full Implementation.

Validates data flow integrity for method chains.
Discrete scoring system: {1.0, 0.8, 0.6, 0.3, 0.0}
"""
import logging
from typing import List, Dict, Any

logger = logging.getLogger(__name__)


class ChainLayerEvaluator:
    """
    Validates chain integrity for method execution.

    Attributes:
        signatures: Dictionary mapping method IDs to their input/output signatures
    """

    def __init__(self, method_signatures: Dict[str, Any]):
        """
        Initialize evaluator with method signatures.

        Args:
            method_signatures: Dict with method input/output signatures
        """
        self.signatures = method_signatures
        logger.info("chain_evaluator_initialized", extra={"num_methods": len(method_signatures)})

    def evaluate(
        self,
        method_id: str,
        provided_inputs: List[str],
        upstream_outputs: Dict[str, str] = None
    ) -> float:
        """
        Validate chain integrity for a method.

```

```
        Discrete scoring: {1.0, 0.8, 0.6, 0.3, 0.0}

        Args:
            method_id: Method to validate
            provided_inputs: Inputs being provided
            upstream_outputs: Types from upstream (for type checking)

        Returns:
            Chain score â\210\210 {0.0, 0.3, 0.6, 0.8, 1.0}
        """
        if method_id not in self.signatures:
            logger.warning("method_signature_missing", extra={"method": method_id})
            return 0.0  # Undeclared method

        sig = self.signatures[method_id]
        required = set(sig.get("required_inputs", []))
        optional = set(sig.get("optional_inputs", []))
        critical_optional = set(sig.get("critical_optional", []))
        provided = set(provided_inputs)

        logger.info(
            "chain_validation_start",
            extra={
                "method": method_id,
                "required": list(required),
                "provided": list(provided)
            }
        )

        # Check 1: Required inputs (HARD FAILURE if missing)
        missing_required = required - provided
        if missing_required:
            logger.error(
                "chain_hard_mismatch",
                extra={
                    "method": method_id,
                    "missing_required": list(missing_required)
                }
            )
            return 0.0  # Hard mismatch

        # Check 2: Critical optional inputs
        missing_critical = critical_optional - provided
        if missing_critical:
            logger.warning(
                "chain_missing_critical_optional",
                extra={
                    "method": method_id,
                    "missing": list(missing_critical)
                }
            )
            return 0.3  # Missing critical optional

        # Check 3: Regular optional inputs
        missing_optional = (optional - critical_optional) - provided
        if missing_optional:
```

```
1121:             logger.info(
1122:                 "chain_missing_optional",
1123:                 extra={
1124:                     "method": method_id,
1125:                     "missing": list(missing_optional)
1126:                 }
1127:             )
1128:             # Check severity: if many missing, lower score
1129:             optional_count = len(optional - critical_optional)
1130:             missing_count = len(missing_optional)
1131:             if optional_count > 0:
1132:                 ratio = missing_count / optional_count
1133:                 if ratio > 0.5:
1134:                     return 0.6  # Many optional missing
1135:                 else:
1136:                     return 0.8  # Some optional missing
1137: 
1138:         # All inputs present
1139:         logger.info("chain_valid", extra={"method": method_id, "score": 1.0})
1140:         return 1.0
1141: 
1142:     def validate_chain_sequence(
1143:         self,
1144:         method_sequence: List[str],
1145:         initial_inputs: List[str]
1146:     ) -> Dict[str, float]:
1147:         """
1148:         Validate entire chain of methods.
1149: 
1150:         Args:
1151:             method_sequence: Ordered list of methods
1152:             initial_inputs: Inputs available at start
1153: 
1154:         Returns:
1155:             Dict mapping method_id to chain score
1156:         """
1157:         results = {}
1158:         available_inputs = set(initial_inputs)
1159: 
1160:         for method_id in method_sequence:
1161:             # Validate this method
1162:             score = self.evaluate(method_id, list(available_inputs))
1163:             results[method_id] = score
1164: 
1165:             # Add this method's output to available inputs
1166:             # (simplified - assumes output name matches method)
1167:             available_inputs.add(f"{method_id}_output")
1168: 
1169:         return results
1170: 
1171:     def compute_chain_quality(
1172:         self,
1173:         method_scores: Dict[str, float]
1174:     ) -> float:
1175:         """
1176:         Compute overall chain quality.
```

```
1177:
1178:         Formula: Minimum score in chain (weakest link)
1179:
1180:         Returns:
1181:             Overall quality â\210\210 [0.0, 1.0]
1182:         """
1183:         if not method_scores:
1184:             return 0.0
1185:
1186:         # Weakest link principle
1187:         min_score = min(method_scores.values())
1188:
1189:         logger.info(
1190:             "chain_quality_computed",
1191:             extra={
1192:                 "min_score": min_score,
1193:                 "num_methods": len(method_scores)
1194:             }
1195:         )
1196:
1197:         return min_score
1198:
1199:
1200:
1201: ================================================================================
1202: FILE: src/farfan_pipeline/core/calibration/choquet_aggregator.py
1203: ================================================================================
1204:
1205: """
1206: Choquet 2-Additive Aggregation.
1207:
1208: This implements the final score computation:
1209:     Cal(I) = Î£ a_â\204\223Â•x_â\204\223 + Î£ a_â\204\223kÂ•min(x_â\204\223, x_k)
1210:
1211: Where:
1212: - First sum: linear terms (weighted sum of layer scores)
1213: - Second sum: interaction terms (synergies via weakest link)
1214: """
1215: import logging
1216: from typing import Dict
1217:
1218: from farfan_pipeline.core.calibration.data_structures import (
1219:     CalibrationResult,
1220:     CalibrationSubject,
1221:     InteractionTerm,
1222:     LayerID,
1223:     LayerScore,
1224: )
1225: from farfan_pipeline.core.calibration.config import ChoquetAggregationConfig
1226:
1227: logger = logging.getLogger(__name__)
1228:
1229:
1230: class ChoquetAggregator:
1231:     """
1232:     Choquet 2-Additive integral aggregator.
```

```
1233:
1234:     This is the FINAL step in the calibration pipeline.
1235:     """
1236:
1237:     def __init__(self, config: ChoquetAggregationConfig):
1238:         self.config = config
1239:
1240:         # Pre-build interaction terms for efficiency
1241:         self.interaction_terms = [
1242:             InteractionTerm(
1243:                 layer_1=LayerID(l1),
1244:                 layer_2=LayerID(l2),
1245:                 weight=weight,
1246:                 rationale=self.config.interaction_rationales.get((l1, l2), "")
1247:             )
1248:             for (l1, l2), weight in self.config.interaction_weights.items()
1249:         ]
1250:
1251:         logger.info(
1252:             "choquet_aggregator_initialized",
1253:             extra={
1254:                 "num_layers": len(self.config.linear_weights),
1255:                 "num_interactions": len(self.interaction_terms),
1256:                 "config_hash": self.config.compute_hash()
1257:             }
1258:         )
1259:
1260:     def aggregate(
1261:         self,
1262:         subject: CalibrationSubject,
1263:         layer_scores: Dict[LayerID, LayerScore],
1264:         metadata: dict = None
1265:     ) -> CalibrationResult:
1266:         """
1267:         Compute final calibration score using Choquet aggregation.
1268:
1269:         Args:
1270:             subject: The calibration subject I = (M, v, Î\223, G, ctx)
1271:             layer_scores: Dict mapping LayerID to LayerScore
1272:             metadata: Additional computation metadata
1273:
1274:         Returns:
1275:             CalibrationResult with final score and full breakdown
1276:
1277:         Raises:
1278:             ValueError: If layer scores are incomplete or invalid
1279:         """
1280:         if metadata is None:
1281:             metadata = {}
1282:
1283:         # Extract numeric scores for computation
1284:         scores = {
1285:             layer_id: layer_score.score
1286:             for layer_id, layer_score in layer_scores.items()
1287:         }
1288:
```

```
        # Verify all expected layers present
        expected_layers = set(LayerID(k) for k in self.config.linear_weights.keys())
        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: Î£ a_â\204\223 Â• x_â\204\223
        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: Î£ a_â\204\223k Â• min(x_â\204\223, 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}_{subject.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/farfan_pipeline/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 Dict

from farfan_pipeline.core.calibration.data_structures import CompatibilityMapping

logger = logging.getLogger(__name__)
```

```
1457:
1458:
1459: class CompatibilityRegistry:
1460:     """
1461:     Registry of method compatibility mappings.
1462:
1463:     This loads from a JSON file that defines which methods work
1464:     in which contexts (questions, dimensions, policies).
1465:     """
1466:
1467:     def __init__(self, config_path: Path | str):
1468:         """
1469:         Initialize registry from configuration file.
1470:
1471:         Args:
1472:             config_path: Path to method_compatibility.json
1473:         """
1474:         self.config_path = Path(config_path)
1475:         self.mappings: Dict[str, CompatibilityMapping] = {}
1476:         self._load()
1477:
1478:     def _load(self):
1479:         """Load compatibility mappings from JSON."""
1480:         if not self.config_path.exists():
1481:             raise FileNotFoundError(
1482:                 f"Compatibility config not found: {self.config_path}\n"
1483:                 f"Create this file with method compatibility definitions."
1484:             )
1485:
1486:         with open(self.config_path, 'r', encoding='utf-8') as f:
1487:             data = json.load(f)
1488:
1489:         # Validate structure
1490:         if "method_compatibility" not in data:
1491:             raise ValueError(
1492:                 "Config must have 'method_compatibility' key at top level"
1493:             )
1494:
1495:         # Load each method's compatibility
1496:         for method_id, compat_data in data["method_compatibility"].items():
1497:             self.mappings[method_id] = CompatibilityMapping(
1498:                 method_id=method_id,
1499:                 questions=compat_data.get("questions", {}),
1500:                 dimensions=compat_data.get("dimensions", {}),
1501:                 policies=compat_data.get("policies", {}),
1502:             )
1503:
1504:             logger.info(
1505:                 "compatibility_loaded",
1506:                 extra={
1507:                     "method": method_id,
1508:                     "num_questions": len(compat_data.get("questions", {})),
1509:                     "num_dimensions": len(compat_data.get("dimensions", {})),
1510:                     "num_policies": len(compat_data.get("policies", {})),
1511:                 }
1512:             )
```

```
1513:
1514:         logger.info(
1515:             "compatibility_registry_loaded",
1516:             extra={"total_methods": len(self.mappings)}
1517:         )
1518:
1519:     def get(self, method_id: str) -> CompatibilityMapping:
1520:         """
1521:         Get compatibility mapping for a method.
1522:
1523:         If method not found, returns a mapping with all penalties (0.1).
1524:         """
1525:         if method_id not in self.mappings:
1526:             logger.warning(
1527:                 "method_compatibility_not_found",
1528:                 extra={
1529:                     "method": method_id,
1530:                     "default_score": 0.1
1531:                 }
1532:             )
1533:             # Return empty mapping (will default to 0.1 penalties)
1534:             return CompatibilityMapping(
1535:                 method_id=method_id,
1536:                 questions={},
1537:                 dimensions={},
1538:                 policies={},
1539:             )
1540:
1541:         return self.mappings[method_id]
1542:
1543:     def validate_anti_universality(self, threshold: float = 0.9) -> dict[str, bool]:
1544:         """
1545:         Check Anti-Universality Theorem for all methods.
1546:
1547:         Returns:
1548:             Dict mapping method_id to compliance status (True = compliant)
1549:
1550:         Raises:
1551:             ValueError if any method violates the theorem
1552:         """
1553:         results = {}
1554:         violations = []
1555:
1556:         for method_id, mapping in self.mappings.items():
1557:             is_compliant = mapping.check_anti_universality(threshold)
1558:             results[method_id] = is_compliant
1559:
1560:             if not is_compliant:
1561:                 violations.append(method_id)
1562:                 logger.error(
1563:                     "anti_universality_violation",
1564:                     extra={
1565:                         "method": method_id,
1566:                         "avg_q": sum(mapping.questions.values()) / len(mapping.questions) if mapping.questions else 0,
1567:                         "avg_d": sum(mapping.dimensions.values()) / len(mapping.dimensions) if mapping.dimensions else 0,
1568:                         "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 â\211¥ {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):
        self.registry = registry

    def evaluate_question(self, method_id: str, question_id: str) -> float:
        """
        Evaluate @q (question compatibility).

        Returns score â\210\210 {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(self, method_id: str, dimension: str) -> float:
        """
        Evaluate @d (dimension compatibility).

        Returns score â\210\210 {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(self, method_id: str, policy_area: str) -> float:
        """
        Evaluate @p (policy area compatibility).

        Returns score â\210\210 {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

    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(method_id, question_id),
            'd': self.evaluate_dimension(method_id, dimension),
            'p': self.evaluate_policy(method_id, policy_area),
        }



================================================================================
FILE: src/farfan_pipeline/core/calibration/config 2.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
"""
from dataclasses import dataclass, field
from typing import Literal
import hashlib
import json
import os


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

    Theoretical Formula:
        U(pdt) = aggregator(w_SÂ•S, w_MÂ•M, w_IÂ•I, w_PÂ•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 â\206\222 U = 0.0)
    # ========================================
    require_ppi_presence: bool = True
    require_indicator_matrix: bool = True
    min_structural_compliance: float = 0.5  # S must be >= this or U = 0.0

    # ========================================
    # Anti-Gaming Thresholds
    # ========================================
    max_placeholder_ratio: float = 0.10  # Max proportion of "S/D" placeholders
    min_unique_values_ratio: float = 0.5  # Min proportion of unique cost values in PPI
    min_number_density: float = 0.02  # Min proportion of numeric tokens in critical sections
```

```
1737:     gaming_penalty_cap: float = 0.3  # Maximum total penalty
1738:
1739:     # ========================================
1740:     # S (Structural Compliance) Sub-Weights
1741:     # ========================================
1742:     w_block_coverage: float = 0.5   # Block presence/quality
1743:     w_hierarchy: float = 0.25       # Header numbering validity
1744:     w_order: float = 0.25           # Sequential order correctness
1745:
1746:     # Block requirements
1747:     min_block_tokens: int = 50      # Min tokens for block to count as "present"
1748:     min_block_numbers: int = 1      # Min numbers for block validity
1749:
1750:     # Hierarchy thresholds
1751:     hierarchy_excellent_threshold: float = 0.8  # â\211¥80% valid â\206\222 score 1.0
1752:     hierarchy_acceptable_threshold: float = 0.5  # â\211¥50% valid â\206\222 score 0.5
1753:
1754:     # ========================================
1755:     # M (Mandatory Sections) Requirements
1756:     # ========================================
1757:     # Section-specific minimums (can be overridden per section type)
1758:     diagnostico_min_tokens: int = 500
1759:     diagnostico_min_keywords: int = 3
1760:     diagnostico_min_numbers: int = 5
1761:     diagnostico_min_sources: int = 2
1762:
1763:     estrategica_min_tokens: int = 400
1764:     estrategica_min_keywords: int = 3
1765:     estrategica_min_numbers: int = 3
1766:
1767:     ppi_section_min_tokens: int = 300
1768:     ppi_section_min_keywords: int = 2
1769:     ppi_section_min_numbers: int = 10
1770:
1771:     seguimiento_min_tokens: int = 200
1772:     seguimiento_min_keywords: int = 2
1773:     seguimiento_min_numbers: int = 2
1774:
1775:     marco_normativo_min_tokens: int = 150
1776:     marco_normativo_min_keywords: int = 1
1777:
1778:     # Critical sections get double weight
1779:     critical_sections_weight: float = 2.0
1780:
1781:     # ========================================
1782:     # I (Indicator Quality) Configuration
1783:     # ========================================
1784:     w_i_struct: float = 0.4   # Structure/completeness
1785:     w_i_link: float = 0.3     # Traceability
1786:     w_i_logic: float = 0.3    # Chain coherence
1787:
1788:     # Hard gate for indicator structure
1789:     i_struct_hard_gate: float = 0.7  # If I_struct < this, I = 0.0
1790:
1791:     # Structure sub-parameters
1792:     i_critical_fields_weight: float = 2.0  # Weight for critical fields
```

```
    i_placeholder_penalty_multiplier: float = 3.0  # Penalty multiplier for "S/D"

    # Link sub-parameters
    i_fuzzy_match_threshold: float = 0.85  # Levenshtein threshold for traceability
    i_mga_code_pattern: str = r"^\d{7}$"  # Valid MGA code format

    # Logic sub-parameters
    i_valid_lb_year_min: int = 2019  # Min valid baseline year
    i_valid_lb_year_max: int = 2024  # Max valid baseline year
    i_valid_meta_year_min: int = 2024  # Min valid target year
    i_valid_meta_year_max: int = 2027  # Max valid target year

    # ========================================
    # P (PPI Completeness) Configuration
    # ========================================
    w_p_presence: float = 0.2      # Matrix exists
    w_p_structure: float = 0.4     # Field completeness
    w_p_consistency: float = 0.4   # Accounting closure

    # Hard gate for PPI structure
    p_struct_hard_gate: float = 0.7  # If P_struct < this, P = 0.0

    # Structure requirements
    p_min_nonzero_rows: float = 0.8  # Min proportion of rows with non-zero costs

    # Consistency tolerances
    p_accounting_tolerance: float = 0.01  # 1% tolerance for sum checks
    p_traceability_threshold: float = 0.80  # Min 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_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_transpÂ•m_transp + w_govÂ•m_gov + w_costÂ•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 (seconds)
    threshold_fast: float = 1.0  # Excellent performance
    threshold_acceptable: float = 5.0  # Acceptable performance
    # If runtime > threshold_acceptable â\206\222 score drops to 0.5
    # If timeout or out_of_memory â\206\222 score = 0.0

    # Memory thresholds (MB)
    threshold_memory_normal: int = 512
    threshold_memory_high: int = 1024

    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) = Î£ a_â\204\223Â•x_â\204\223 + Î£ a_â\204\223kÂ•min(x_â\204\223, x_k)

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

    Mathematical Constraint:
        Î£ a_â\204\223 + Î£ a_â\204\223k = 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:
    # Î£ a_â\204\223 + Î£ a_â\204\223k = 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_â\204\223, x_k) "weakest link" principle
    interaction_weights: dict[tuple[str, str], float] = field(default_factory=lambda: {
        ("u", "chain"): 0.15,   # Plan quality Ã\227 Chain integrity
        ("chain", "C"): 0.12,   # Chain integrity Ã\227 Congruence
        ("q", "d"): 0.08,       # Question Ã\227 Dimension
        ("d", "p"): 0.05,       # Dimension Ã\227 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}")

    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  # Threshold for universality detection

    # Determinism (SIN_CARRETA doctrine)
    random_seed: int = 42
    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,
        }
```

```
2129:         config_json = json.dumps(config_dict, sort_keys=True, separators=(',', ':'))
2130:         return hashlib.sha256(config_json.encode('utf-8')).hexdigest()
2131:
2132:     def to_dict(self) -> dict:
2133:         """Export configuration as dictionary."""
2134:         return {
2135:             "system_hash": self.compute_system_hash(),
2136:             "unit_layer": {
2137:                 "weights": {
2138:                     "S": self.unit_layer.w_S,
2139:                     "M": self.unit_layer.w_M,
2140:                     "I": self.unit_layer.w_I,
2141:                     "P": self.unit_layer.w_P,
2142:                 },
2143:                 "aggregation_type": self.unit_layer.aggregation_type,
2144:                 "hard_gates": {
2145:                     "require_ppi": self.unit_layer.require_ppi_presence,
2146:                     "require_indicators": self.unit_layer.require_indicator_matrix,
2147:                     "min_structural": self.unit_layer.min_structural_compliance,
2148:                 }
2149:             },
2150:             "meta_layer": {
2151:                 "weights": {
2152:                     "transparency": self.meta_layer.w_transparency,
2153:                     "governance": self.meta_layer.w_governance,
2154:                     "cost": self.meta_layer.w_cost,
2155:                 }
2156:             },
2157:             "choquet": {
2158:                 "linear_weights": self.choquet.linear_weights,
2159:                 "interaction_count": len(self.choquet.interaction_weights),
2160:                 "config_hash": self.choquet.compute_hash(),
2161:             },
2162:             "system": {
2163:                 "anti_universality_enabled": self.enable_anti_universality_check,
2164:                 "deterministic": self.enforce_determinism,
2165:                 "random_seed": self.random_seed,
2166:             }
2167:         }
2168:
2169:
2170: # ======================================
2171: # Default Configuration Instance
2172: # ======================================
2173: DEFAULT_CALIBRATION_CONFIG = CalibrationSystemConfig()
2174:
2175:
2176:
2177: ================================================================================
2178: FILE: src/farfan_pipeline/core/calibration/config.py
2179: ================================================================================
2180:
2181: """Calibration config - LEGACY STUB for backward compatibility."""
2182:
2183: DEFAULT_CALIBRATION_CONFIG: dict[str, object] = {}
2184:
```

```
2185:
2186:
2187: ================================================================================
2188: FILE: src/farfan_pipeline/core/calibration/congruence_layer.py
2189: ================================================================================
2190:
2191: """
2192: Congruence Layer (@C) - Full Implementation.
2193:
2194: Evaluates method ensemble congruence using three components:
2195: - c_scale: Range compatibility
2196: - c_sem: Semantic overlap (Jaccard index)
2197: - c_fusion: Fusion rule validity
2198:
2199: Formula: C_play(G|ctx) = c_scale Â• c_sem Â• c_fusion
2200: """
2201: import logging
2202: from typing import List, Dict, Any
2203:
2204: logger = logging.getLogger(__name__)
2205:
2206:
2207: class CongruenceLayerEvaluator:
2208:     """
2209:     Evaluates congruence of method ensembles.
2210:
2211:     Attributes:
2212:         registry: Dictionary mapping method IDs to their metadata
2213:     """
2214:
2215:     def __init__(self, method_registry: Dict[str, Any]):
2216:         """
2217:         Initialize evaluator with method registry.
2218:
2219:         Args:
2220:             method_registry: Dict with method metadata (output_range, semantic_tags, etc.)
2221:         """
2222:         self.registry = method_registry
2223:         logger.info("congruence_evaluator_initialized", extra={"num_methods": len(method_registry)})
2224:
2225:     def evaluate(
2226:         self,
2227:         method_ids: List[str],
2228:         subgraph_id: str,
2229:         fusion_rule: str = "weighted_average",
2230:         provided_inputs: List[str] = None
2231:     ) -> float:
2232:         """
2233:         Evaluate congruence of method ensemble.
2234:
2235:         Formula: C_play(G|ctx) = c_scale Â• c_sem Â• c_fusion
2236:
2237:         Args:
2238:             method_ids: List of methods in the subgraph
2239:             subgraph_id: Identifier for this subgraph
2240:             fusion_rule: How outputs are combined (weighted_average, max, min, product)
```

```
            provided_inputs: List of actual inputs provided

        Returns:
            C_play â\210\210 [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:
        """
```

```
2297:         Compute c_scale: Range compatibility.
2298:
2299:         Scoring:
2300:             1.0: All ranges identical
2301:             0.8: All ranges convertible (within [0,1])
2302:             0.0: Incompatible ranges
2303:
2304:         Returns:
2305:             c_scale â\210\210 {0.0, 0.8, 1.0}
2306:         """
2307:         ranges = []
2308:         for method_id in method_ids:
2309:             if method_id not in self.registry:
2310:                 logger.warning("method_not_registered", extra={"method": method_id})
2311:                 return 0.0
2312:
2313:             method_data = self.registry[method_id]
2314:             output_range = method_data.get("output_range")
2315:
2316:             if output_range is None:
2317:                 logger.warning("no_output_range", extra={"method": method_id})
2318:                 return 0.0
2319:
2320:             ranges.append(tuple(output_range))
2321:
2322:         # Check if all ranges are identical
2323:         first_range = ranges[0]
2324:         if all(r == first_range for r in ranges):
2325:             logger.debug("ranges_identical", extra={"range": first_range})
2326:             return 1.0
2327:
2328:         # Check if all ranges are in [0,1] (convertible)
2329:         all_in_unit = all(r == (0.0, 1.0) for r in ranges)
2330:         if all_in_unit:
2331:             logger.debug("ranges_convertible", extra={"note": "all in [0,1]"})
2332:             return 0.8
2333:
2334:         # Incompatible ranges
2335:         logger.warning("ranges_incompatible", extra={"ranges": ranges})
2336:         return 0.0
2337:
2338:     def _compute_semantic_congruence(self, method_ids: List[str]) -> float:
2339:         """
2340:         Compute c_sem: Semantic overlap (Jaccard index).
2341:
2342:         Formula: |intersection| / |union| of semantic tags
2343:
2344:         Returns:
2345:             c_sem â\210\210 [0.0, 1.0]
2346:         """
2347:         tag_sets = []
2348:
2349:         for method_id in method_ids:
2350:             if method_id not in self.registry:
2351:                 logger.warning("method_not_registered", extra={"method": method_id})
2352:                 return 0.0
```

```
2353:
2354:             tags = self.registry[method_id].get("semantic_tags", [])
2355:             if not isinstance(tags, (list, set)):
2356:                 tags = []
2357:             tag_sets.append(set(tags))
2358:
2359:         if not tag_sets:
2360:             return 0.0
2361:
2362:         # Compute intersection and union
2363:         intersection = tag_sets[0]
2364:         union = tag_sets[0]
2365:
2366:         for tags in tag_sets[1:]:
2367:             intersection = intersection.intersection(tags)
2368:             union = union.union(tags)
2369:
2370:         # Jaccard index
2371:         if len(union) == 0:
2372:             logger.warning("no_semantic_tags", extra={"methods": method_ids})
2373:             return 0.0
2374:
2375:         jaccard = len(intersection) / len(union)
2376:
2377:         logger.debug(
2378:             "semantic_congruence",
2379:             extra={
2380:                 "intersection": list(intersection),
2381:                 "union_size": len(union),
2382:                 "jaccard": jaccard
2383:             }
2384:         )
2385:
2386:         return jaccard
2387:
2388:     def _compute_fusion_validity(
2389:         self,
2390:         method_ids: List[str],
2391:         fusion_rule: str,
2392:         provided_inputs: List[str]
2393:     ) -> float:
2394:         """
2395:         Compute c_fusion: Fusion rule validity.
2396:
2397:         Scoring:
2398:             1.0: Rule valid AND all inputs present
2399:             0.5: Rule valid BUT some inputs missing
2400:             0.0: Rule invalid
2401:
2402:         Returns:
2403:             c_fusion â\210\210 {0.0, 0.5, 1.0}
2404:         """
2405:         # Check if fusion rule is valid
2406:         valid_rules = ["weighted_average", "max", "min", "product", "custom"]
2407:         if fusion_rule not in valid_rules:
2408:             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


```