```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 22
================================================================================
Generated: 2025-12-07T06:17:28.392472
Files in this batch: 17
================================================================================


================================================================================
FILE: src/farfan_pipeline/utils/json_logger.py
================================================================================

"""
Lightweight JSON Logging - Structured Event Logging
================================================

Provides structured JSON logging for the pipeline with:
- JSON formatter for LogRecord
- Helper for logging I/O events with envelope metadata
- No PII logging
- Correlation ID and event ID tracking

Author: Policy Analytics Research Unit
Version: 1.0.0
License: Proprietary
"""

from __future__ import annotations

import json
import logging
import time
from typing import Any
from farfan_pipeline.core.calibration.decorators import calibrated_method

# Import will be available at runtime
try:
    from farfan_pipeline.utils.contract_io import ContractEnvelope
except ImportError:
    # Allow module to load for testing
    ContractEnvelope = None  # type: ignore


class JsonFormatter(logging.Formatter):
    """
    JSON formatter for structured logging.

    Formats LogRecord as JSON with standard fields plus custom extras.
    """

    @calibrated_method("farfan_core.utils.json_logger.JsonFormatter.format")
    def format(self, record: logging.LogRecord) -> str:
        """
        Format LogRecord as JSON string.

        Args:
```

```
            record: LogRecord to format

        Returns:
            JSON string representation
        """
        payload: dict[str, Any] = {
            "level": record.levelname,
            "logger": record.name,
            "message": record.getMessage(),
            "timestamp_utc": record.__dict__.get("timestamp_utc"),
            "event_id": record.__dict__.get("event_id"),
            "correlation_id": record.__dict__.get("correlation_id"),
            "policy_unit_id": record.__dict__.get("policy_unit_id"),
            "phase": record.__dict__.get("phase"),
            "latency_ms": record.__dict__.get("latency_ms"),
            "input_bytes": record.__dict__.get("input_bytes"),
            "output_bytes": record.__dict__.get("output_bytes"),
            "input_digest": record.__dict__.get("input_digest"),
            "output_digest": record.__dict__.get("output_digest"),
        }
        # Drop None values to keep JSON compact
        payload = {k: v for k, v in payload.items() if v is not None}
        return json.dumps(payload, separators=(",", ":"), ensure_ascii=False)


def get_json_logger(name: str = "farfan_core") -> logging.Logger:
    """
    Get or create a JSON logger.

    Creates a logger with JSON formatting if not already configured.

    Args:
        name: Logger name

    Returns:
        Configured logger instance

    Examples:
        >>> logger = get_json_logger("test")
        >>> logger.name
        'test'
        >>> logger.level
        20
    """
    logger = logging.getLogger(name)
    if not any(isinstance(h, logging.StreamHandler) for h in logger.handlers):
        h = logging.StreamHandler()
        h.setFormatter(JsonFormatter())
        logger.addHandler(h)
        logger.setLevel(logging.INFO)
        logger.propagate = False
    return logger


def log_io_event(
    logger: logging.Logger,
```

```
    *,
    phase: str,
    envelope_in: Any | None,  # ContractEnvelope or None
    envelope_out: Any,  # ContractEnvelope
    started_monotonic: float,
) -> None:
    """
    Log an I/O event with envelope metadata.

    Args:
        logger: Logger instance
        phase: Phase name
        envelope_in: Input envelope (may be None)
        envelope_out: Output envelope
        started_monotonic: Monotonic start time

    Examples:
        >>> import time
        >>> from farfan_core.utils.contract_io import ContractEnvelope
        >>> logger = get_json_logger("test")
        >>> out = ContractEnvelope.wrap(
        ...     {"ok": True},
        ...     policy_unit_id="PU_123",
        ...     correlation_id="corr-1"
        ... )
        >>> # This will log JSON to stdout
        >>> log_io_event(
        ...     logger,
        ...     phase="normalize",
        ...     envelope_in=None,
        ...     envelope_out=out,
        ...     started_monotonic=time.monotonic()
        ... )  # doctest: +SKIP
    """
    elapsed_ms = int((time.monotonic() - started_monotonic) * 1000)

    # Safely get payload sizes
    input_bytes = None
    if envelope_in is not None:
        try:
            payload = getattr(envelope_in, "payload", None)
            if payload is not None:
                input_bytes = len(json.dumps(payload, ensure_ascii=False))
        except (TypeError, AttributeError):
            pass

    output_bytes = None
    try:
        output_bytes = len(json.dumps(envelope_out.payload, ensure_ascii=False))
    except (TypeError, AttributeError):
        # If payload is missing or not serializable, skip logging output_bytes.
        # This is non-critical for logging; output_bytes will be None.
        pass

    logger.info(
        "phase_io",
```

```
169:         extra={
170:             "timestamp_utc": envelope_out.timestamp_utc,
171:             "event_id": envelope_out.event_id,
172:             "correlation_id": envelope_out.correlation_id,
173:             "policy_unit_id": envelope_out.policy_unit_id,
174:             "phase": phase,
175:             "latency_ms": elapsed_ms,
176:             "input_bytes": input_bytes,
177:             "output_bytes": output_bytes,
178:             "input_digest": getattr(envelope_in, "content_digest", None),
179:             "output_digest": envelope_out.content_digest,
180:         },
181:     )
182: 
183: 
184: if __name__ == "__main__":
185:     import doctest
186:     import time
187: 
188:     # Run doctests
189:     print("Running doctests...")
190:     doctest.testmod(verbose=True)
191: 
192:     # Integration tests
193:     print("\n" + "="*60)
194:     print("JSON Logger Integration Tests")
195:     print("="*60)
196: 
197:     print("\n1. Testing JSON formatter:")
198:     logger = get_json_logger("demo")
199:     assert logger.level == logging.INFO
200:     assert len(logger.handlers) > 0
201:     assert isinstance(logger.handlers[0].formatter, JsonFormatter)
202:     print("   â\234\223 Logger configured with JSON formatter")
203: 
204:     print("\n2. Testing log output structure:")
205:     # Create a test record
206:     record = logging.LogRecord(
207:         name="test",
208:         level=logging.INFO,
209:         pathname="",
210:         lineno=0,
211:         msg="test message",
212:         args=(),
213:         exc_info=None,
214:     )
215:     record.event_id = "evt-123"
216:     record.correlation_id = "corr-456"
217:     record.latency_ms = 42
218: 
219:     formatter = JsonFormatter()
220:     output = formatter.format(record)
221:     parsed = json.loads(output)
222: 
223:     assert parsed["level"] == "INFO"
224:     assert parsed["message"] == "test message"
```

```
    assert parsed["event_id"] == "evt-123"
    assert parsed["correlation_id"] == "corr-456"
    assert parsed["latency_ms"] == 42
    print("   â\234\223 JSON format includes all expected fields")

    print("\n3. Testing I/O event logging:")
    # Only test if ContractEnvelope is available
    if ContractEnvelope is not None:
        from farfan_pipeline.utils.contract_io import ContractEnvelope

        lg = get_json_logger("demo")
        out = ContractEnvelope.wrap(
            {"ok": True},
            policy_unit_id="PU_123",
            correlation_id="corr-1"
        )

        # Capture the log output
        import io
        import sys
        old_stdout = sys.stdout
        sys.stdout = buffer = io.StringIO()

        log_io_event(
            lg,
            phase="normalize",
            envelope_in=None,
            envelope_out=out,
            started_monotonic=time.monotonic()
        )

        sys.stdout = old_stdout
        log_output = buffer.getvalue()

        # Verify JSON output
        if log_output.strip():
            log_data = json.loads(log_output.strip())
            assert log_data["phase"] == "normalize"
            assert log_data["policy_unit_id"] == "PU_123"
            assert "latency_ms" in log_data
            print("   â\234\223 I/O event logged with correct structure")
        else:
            print("   â\234\223 I/O event logging executed (output suppressed)")
    else:
        print("   â\212\230 Skipped (ContractEnvelope not available)")

    print("\n" + "="*60)
    print("JSON logger doctest OK - All tests passed!")
    print("="*60)



================================================================================
FILE: src/farfan_pipeline/utils/metadata_loader.py
================================================================================

```

```
"""
Metadata Loader with Supply-Chain Security
Implements fail-fast validation with version pinning, checksum verification, and schema validation
"""

import hashlib
import json
import logging
from pathlib import Path
from typing import Any

import yaml

from farfan_pipeline.utils.paths import proj_root
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method

try:
    import jsonschema
    JSONSCHEMA_AVAILABLE = True
except ImportError:
    JSONSCHEMA_AVAILABLE = False
    logging.warning("jsonschema not available - schema validation disabled")

logger = logging.getLogger(__name__)

class MetadataError(Exception):
    """Base exception for metadata errors"""
    pass

class MetadataVersionError(MetadataError):
    """Version mismatch error"""
    def __init__(self, expected: str, actual: str, file_path: str) -> None:
        self.expected = expected
        self.actual = actual
        self.file_path = file_path
        super().__init__(
            f"Version mismatch in {file_path}: expected {expected}, got {actual}"
        )

class MetadataIntegrityError(MetadataError):
    """Checksum/integrity violation error"""
    def __init__(self, file_path: str, expected_checksum: str | None = None, actual_checksum: str | None = None) -> None:
        self.file_path = file_path
        self.expected_checksum = expected_checksum
        self.actual_checksum = actual_checksum
        msg = f"Integrity violation in {file_path}"
        if expected_checksum and actual_checksum:
            msg += f": expected checksum {expected_checksum}, got {actual_checksum}"
        super().__init__(msg)

class MetadataSchemaError(MetadataError):
    """Schema validation error"""
    def __init__(self, file_path: str, validation_errors: list) -> None:
        self.file_path = file_path
        self.validation_errors = validation_errors
```

```
337:         error_msgs = '\n'.join(f"  - {err}" for err in validation_errors)
338:         super().__init__(
339:             f"Schema validation failed for {file_path}:\n{error_msgs}"
340:         )
341:
342: class MetadataMissingKeyError(MetadataError):
343:     """Required key missing in metadata"""
344:     def __init__(self, file_path: str, missing_key: str, context: str = "") -> None:
345:         self.file_path = file_path
346:         self.missing_key = missing_key
347:         self.context = context
348:         msg = f"Required key '{missing_key}' missing in {file_path}"
349:         if context:
350:             msg += f" ({context})"
351:         super().__init__(msg)
352:
353: class MetadataLoader:
354:     """
355:     Unified metadata loader with strict validation
356:
357:     Features:
358:     - Version pinning with semantic versioning
359:     - SHA-256 checksum verification
360:     - JSON Schema validation
361:     - Fail-fast on any violation
362:     - Structured logging of all errors
363:     """
364:
365:     def __init__(self, workspace_root: Path | None = None) -> None:
366:         self.workspace_root = Path(workspace_root) if workspace_root else proj_root()
367:         self.schemas_dir = self.workspace_root / "schemas"
368:
369:         # Loaded schemas cache
370:         self._schema_cache: dict[str, dict] = {}
371:
372:     def load_and_validate_metadata(
373:         self,
374:         path: Path,
375:         schema_ref: str | None = None,
376:         required_version: str | None = None,
377:         expected_checksum: str | None = None,
378:         checksum_algorithm: str = "sha256"
379:     ) -> dict[str, Any]:
380:         """
381:         Load and validate metadata file with all safeguards
382:
383:         Args:
384:             path: Path to metadata file (JSON or YAML)
385:             schema_ref: Schema file name (e.g., "rubric.schema.json")
386:             required_version: Required version string (e.g., "2.ParameterLoaderV2.get("farfan_core.utils.metadata_loader.MetadataLoader.__init__", "auto_par
am_L105_64", 0.0)")
387:             expected_checksum: Expected SHA-256 checksum (hex)
388:             checksum_algorithm: Hash algorithm ("sha256", "md5")
389:
390:         Returns:
391:             Validated metadata dictionary
```

```
392:
393:         Raises:
394:             MetadataVersionError: Version mismatch
395:             MetadataIntegrityError: Checksum mismatch
396:             MetadataSchemaError: Schema validation failure
397:             MetadataMissingKeyError: Required key missing
398:         """
399:
400:         # 1. Load file
401:         metadata = self._load_file(path)
402:
403:         # 2. Version check
404:         if required_version:
405:             actual_version = metadata.get("version")
406:             if not actual_version:
407:                 raise MetadataMissingKeyError(str(path), "version", "version field required")
408:
409:             if actual_version != required_version:
410:                 self._log_error(
411:                     rule_id="VERSION_MISMATCH",
412:                     file_path=str(path),
413:                     expected=required_version,
414:                     actual=actual_version
415:                 )
416:                 raise MetadataVersionError(required_version, actual_version, str(path))
417:
418:             logger.info(f"â\234\223 Version validated: {path.name} v{actual_version}")
419:
420:         # 3. Checksum verification
421:         if expected_checksum:
422:             actual_checksum = self._calculate_checksum(metadata, checksum_algorithm)
423:
424:             if actual_checksum != expected_checksum:
425:                 self._log_error(
426:                     rule_id="CHECKSUM_MISMATCH",
427:                     file_path=str(path),
428:                     expected=expected_checksum,
429:                     actual=actual_checksum
430:                 )
431:                 raise MetadataIntegrityError(str(path), expected_checksum, actual_checksum)
432:
433:             logger.info(f"â\234\223 Checksum validated: {path.name} ({checksum_algorithm})")
434:
435:         # 4. Schema validation
436:         if schema_ref and JSONSCHEMA_AVAILABLE:
437:             schema = self._load_schema(schema_ref)
438:             errors = self._validate_schema(metadata, schema)
439:
440:             if errors:
441:                 self._log_error(
442:                     rule_id="SCHEMA_VALIDATION_FAILED",
443:                     file_path=str(path),
444:                     errors=errors
445:                 )
446:                 raise MetadataSchemaError(str(path), errors)
447:
```

```
448:             logger.info(f"â\234\223 Schema validated: {path.name}")
449:
450:         return metadata
451:
452:     @calibrated_method("farfan_core.utils.metadata_loader.MetadataLoader._load_file")
453:     def _load_file(self, path: Path) -> dict[str, Any]:
454:         """Load JSON or YAML file"""
455:         if not path.exists():
456:             raise FileNotFoundError(f"Metadata file not found: {path}")
457:
458:         try:
459:             with open(path, encoding='utf-8') as f:
460:                 content = f.read()
461:
462:             if path.suffix in ['.json']:
463:                 return json.loads(content)
464:             elif path.suffix in ['.yaml', '.yml']:
465:                 return yaml.safe_load(content)
466:             else:
467:                 raise ValueError(f"Unsupported file type: {path.suffix}")
468:
469:         except (json.JSONDecodeError, yaml.YAMLError) as e:
470:             raise MetadataError(f"Failed to parse {path}: {e}")
471:
472:     @calibrated_method("farfan_core.utils.metadata_loader.MetadataLoader._calculate_checksum")
473:     def _calculate_checksum(self, metadata: dict[str, Any], algorithm: str = "sha256") -> str:
474:         """
475:         Calculate reproducible checksum of metadata
476:
477:         Normalization:
478:         - JSON serialization with sorted keys
479:         - UTF-8 encoding
480:         - No whitespace variations
481:         """
482:         normalized = json.dumps(metadata, sort_keys=True, separators=(',', ':'))
483:
484:         if algorithm == "sha256":
485:             return hashlib.sha256(normalized.encode('utf-8')).hexdigest()
486:         elif algorithm == "md5":
487:             return hashlib.md5(normalized.encode('utf-8')).hexdigest()
488:         else:
489:             raise ValueError(f"Unsupported algorithm: {algorithm}")
490:
491:     @calibrated_method("farfan_core.utils.metadata_loader.MetadataLoader._load_schema")
492:     def _load_schema(self, schema_ref: str) -> dict[str, Any]:
493:         """Load JSON Schema from schemas directory"""
494:         if schema_ref in self._schema_cache:
495:             return self._schema_cache[schema_ref]
496:
497:         schema_path = self.schemas_dir / schema_ref
498:
499:         if not schema_path.exists():
500:             raise FileNotFoundError(f"Schema not found: {schema_path}")
501:
502:         with open(schema_path, encoding='utf-8') as f:
503:             schema = json.load(f)
```

```
504:
505:         self._schema_cache[schema_ref] = schema
506:         return schema
507:
508:     @calibrated_method("farfan_core.utils.metadata_loader.MetadataLoader._validate_schema")
509:     def _validate_schema(self, metadata: dict[str, Any], schema: dict[str, Any]) -> list:
510:         """Validate metadata against JSON Schema"""
511:         if not JSONSCHEMA_AVAILABLE:
512:             logger.warning("jsonschema not available - skipping schema validation")
513:             return []
514:
515:         # Import check for type checker
516:         import jsonschema as js
517:         validator = js.Draft7Validator(schema)
518:         errors = []
519:
520:         for error in validator.iter_errors(metadata):
521:             error_path = '.'.join(str(p) for p in error.path) if error.path else 'root'
522:             errors.append(f"{error_path}: {error.message}")
523:
524:         return errors
525:
526:     @calibrated_method("farfan_core.utils.metadata_loader.MetadataLoader._log_error")
527:     def _log_error(self, rule_id: str, file_path: str, **kwargs) -> None:
528:         """Structured error logging"""
529:         from datetime import datetime, timezone
530:
531:         log_entry = {
532:             "timestamp": datetime.now(timezone.utc).isoformat(),
533:             "level": "ERROR",
534:             "rule_id": rule_id,
535:             "file_path": file_path,
536:             **kwargs
537:         }
538:
539:         logger.error(json.dumps(log_entry, indent=2))
540:
541: # REMOVED: load_cuestionario() - LEGACY FUNCTION
542: # Questionnaire monolith must ONLY be loaded via factory.load_questionnaire_monolith()
543: # This enforces architectural requirement: Single I/O boundary in factory.py
544: # See: src/farfan_core/core/orchestrator/factory.py::load_questionnaire_monolith()
545:
546: def load_execution_mapping(
547:     path: Path | None = None,
548:     required_version: str = "2.0"
549: ) -> dict[str, Any]:
550:     """
551:     Load and validate execution_mapping.yaml
552:
553:     Args:
554:         path: Path to execution mapping (default: execution_mapping.yaml)
555:         required_version: Required version
556:
557:     Returns:
558:         Validated execution mapping
559:     """
```

```
    if path is None:
        path = proj_root() / "execution_mapping.yaml"

    loader = MetadataLoader()
    return loader.load_and_validate_metadata(
        path=path,
        schema_ref="execution_mapping.schema.json",
        required_version=required_version
    )

def load_rubric_scoring(
    path: Path | None = None,
    required_version: str = "2.0"
) -> dict[str, Any]:
    """
    Load and validate rubric_scoring.json

    Args:
        path: Path to rubric scoring (default: rubric_scoring.json)
        required_version: Required version

    Returns:
        Validated rubric scoring configuration
    """
    if path is None:
        path = proj_root() / "rubric_scoring.json"

    loader = MetadataLoader()
    return loader.load_and_validate_metadata(
        path=path,
        schema_ref="rubric.schema.json",
        required_version=required_version
    )



================================================================================
FILE: src/farfan_pipeline/utils/method_config_loader.py
================================================================================

"""
Method Configuration Loader for Canonical JSON Specification.

Provides unified access to method parameters from the canonical
parameterization specification.
"""
import ast
import json
from pathlib import Path
from typing import Any
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method


class MethodConfigLoader:
    """
```

```
    Loads and provides access to method parameters from canonical JSON.

    Usage:
        loader = MethodConfigLoader("CANONICAL_METHOD_PARAMETERIZATION_SPEC.json")
        threshold = loader.get_method_parameter(
            "CAUSAL.BMI.infer_mech_v1",
            "kl_divergence_threshold"
        )

    Note:
        The loader expects the JSON spec to follow the canonical schema with
        keys: specification_metadata, methods, and epistemic_validation_summary.
    """

    def __init__(self, spec_path: str | Path) -> None:
        self.spec_path = Path(spec_path)
        with open(self.spec_path) as f:
            self.spec = json.load(f)

        # Validate schema before use
        self.validate_spec_schema()

        # Build index for fast lookup
        self._method_index = {
            method["canonical_id"]: method
            for method in self.spec["methods"]
        }

    @calibrated_method("farfan_core.utils.method_config_loader.MethodConfigLoader.validate_spec_schema")
    def validate_spec_schema(self) -> None:
        """
        Validate JSON spec matches expected schema.

        Raises:
            ValueError: If spec is missing required keys
        """
        required_keys = {"specification_metadata", "methods"}
        # Note: epistemic_validation_summary is optional in some versions
        if not required_keys.issubset(self.spec.keys()):
            missing = required_keys - set(self.spec.keys())
            raise ValueError(f"Spec missing required keys: {missing}")

    def get_method_parameter(
        self,
        canonical_id: str,
        param_name: str,
        override: Any = None
    ) -> Any:
        """
        Get parameter value for a method.

        Args:
            canonical_id: Canonical method ID (e.g., "CAUSAL.BMI.infer_mech_v1")
            param_name: Parameter name
            override: Optional override value (takes precedence over default)

```

```
        Returns:
            Parameter value (default or override)

        Raises:
            KeyError: If method or parameter not found
        """
        if canonical_id not in self._method_index:
            raise KeyError(f"Method {canonical_id} not found in canonical spec")

        method = self._method_index[canonical_id]

        for param in method["parameters"]:
            if param["name"] == param_name:
                return override if override is not None else param["default"]

        raise KeyError(f"Parameter {param_name} not found for method {canonical_id}")

    @calibrated_method("farfan_core.utils.method_config_loader.MethodConfigLoader.get_method_description")
    def get_method_description(self, canonical_id: str) -> str:
        """Get method description."""
        return self._method_index[canonical_id]["description"]

    @calibrated_method("farfan_core.utils.method_config_loader.MethodConfigLoader.get_parameter_spec")
    def get_parameter_spec(self, canonical_id: str, param_name: str) -> dict:
        """Get full parameter specification including allowed values."""
        method = self._method_index[canonical_id]
        for param in method["parameters"]:
            if param["name"] == param_name:
                return param
        raise KeyError(f"Parameter {param_name} not found")

    def validate_parameter_value(
        self,
        canonical_id: str,
        param_name: str,
        value: Any
    ) -> bool:
        """
        Validate parameter value against allowed_values specification.

        Returns:
            True if valid, raises ValueError if invalid
        """
        param_spec = self.get_parameter_spec(canonical_id, param_name)
        allowed = param_spec["allowed_values"]

        if allowed["kind"] == "range":
            spec = allowed["spec"]
            min_val, max_val = self._parse_range(spec)
            if not (min_val <= value <= max_val):
                raise ValueError(f"{param_name}={value} out of range {spec}")

        elif allowed["kind"] == "set":
            spec = allowed["spec"]
            valid_values = self._parse_set(spec)
            if value not in valid_values:
```

```
728:                 raise ValueError(f"{param_name}={value} not in allowed set {spec}")
729:
730:         return True
731:
732:     @calibrated_method("farfan_core.utils.method_config_loader.MethodConfigLoader._parse_range")
733:     def _parse_range(self, spec: str) -> tuple[float, float]:
734:         """
735:         Parse range specification like '[ParameterLoaderV2.get("farfan_core.utils.method_config_loader.MethodConfigLoader._parse_range", "auto_param_L135_41
", 0.0), ParameterLoaderV2.get("farfan_core.utils.method_config_loader.MethodConfigLoader._parse_range", "auto_param_L135_46", 1.0)], inclusive' or '[100, 10000],
integer'.
736:
737:         Args:
738:             spec: Range specification string with format "[min, max], modifiers"
739:                   Modifiers can include: inclusive, exclusive, integer
740:
741:         Returns:
742:             Tuple of (min_val, max_val) as floats
743:
744:         Raises:
745:             ValueError: If spec format is invalid
746:
747:         Note:
748:             The inclusive/exclusive and integer modifiers are parsed but not
749:             currently enforced in validation. This maintains compatibility with
750:             the current spec while allowing future enhancement.
751:         """
752:         try:
753:             # Extract bracketed part before any modifiers
754:             bracket_part = spec.split("]")[0] + "]"
755:             parts = bracket_part.replace("[", "").replace("]", "").split(",")
756:             min_val = float(parts[0].strip())
757:             max_val = float(parts[1].strip())
758:             return min_val, max_val
759:         except (IndexError, ValueError) as e:
760:             raise ValueError(f"Invalid range spec: {spec}") from e
761:
762:     @calibrated_method("farfan_core.utils.method_config_loader.MethodConfigLoader._parse_set")
763:     def _parse_set(self, spec: str | list) -> set:
764:         """
765:         Parse set specification safely.
766:
767:         Args:
768:             spec: Either a list or a string representation of a Python literal
769:
770:         Returns:
771:             Set of allowed values
772:
773:         Raises:
774:             ValueError: If spec cannot be parsed safely
775:
776:         Note:
777:             Uses ast.literal_eval() for safe parsing of string specs.
778:             Only Python literals (strings, numbers, tuples, lists, dicts,
779:             booleans, None) are supported - no arbitrary code execution.
780:         """
781:         if isinstance(spec, list):
```

```
            return set(spec)
        try:
            # Use ast.literal_eval for safer parsing
            return set(ast.literal_eval(spec))
        except (ValueError, SyntaxError) as e:
            raise ValueError(f"Invalid set spec: {spec}") from e


================================================================================
FILE: src/farfan_pipeline/utils/paths.py
================================================================================

"""
Portable, secure, and deterministic path utilities for SAAAAAA.

This module provides cross-platform path operations that ensure:
- Portability across Linux, macOS, and Windows
- Security through path traversal protection
- Determinism via normalized paths
- Controlled write locations (never in source tree)

All path operations in the repository MUST use these utilities instead of:
- Direct __file__ usage for resource access
- sys.path manipulation
- Hardcoded absolute paths
- os.path functions (use pathlib.Path instead)
"""

from __future__ import annotations

import os
import unicodedata
from pathlib import Path
from typing import Final
from farfan_pipeline.core.calibration.decorators import calibrated_method


# Custom exception types for path errors
class PathError(Exception):
    """Base exception for path-related errors."""
    pass


class PathTraversalError(PathError):
    """Raised when a path attempts to escape workspace boundaries."""
    pass


class PathNotFoundError(PathError):
    """Raised when a required path does not exist."""
    pass


class PathOutsideWorkspaceError(PathError):
    """Raised when a path is outside the allowed workspace."""
```

```
838:     pass
839:
840:
841: class UnnormalizedPathError(PathError):
842:     """Raised when a path is not properly normalized."""
843:     pass
844:
845:
846: # Project root detection - computed once at module load
847: def _detect_project_root() -> Path:
848:     """
849:     Detect the project root directory using filesystem markers.
850:
851:     This function uses a multi-strategy approach to locate the project root:
852:
853:     1. Primary strategy: Search for pyproject.toml
854:        - Walks up the directory tree from this file's location
855:        - Returns the first directory containing pyproject.toml
856:
857:     2. Secondary strategy: Search for src/farfan_core layout
858:        - Looks for directories with both src/farfan_core and setup.py
859:        - This supports older project structures
860:
861:     3. Fallback strategy: Relative path calculation
862:        - If no markers found, assumes standard layout (src/farfan_core/utils)
863:        - Returns path 3 levels up from this file
864:
865:     The function is called once at module load time, and the result is
866:     cached in the PROJECT_ROOT constant.
867:
868:     Returns:
869:         Path: Absolute path to the project root directory
870:
871:     Raises:
872:         No exceptions raised; always returns a path (uses fallback if needed)
873:
874:     Note:
875:         This function is intended for internal use. External code should use
876:         the PROJECT_ROOT constant instead of calling this directly.
877:     """
878:     # Start from this file's location
879:     current = Path(__file__).resolve().parent
880:
881:     # Walk up to find pyproject.toml
882:     for parent in [current] + list(current.parents):
883:         if (parent / "pyproject.toml").exists():
884:             return parent
885:         if (parent / "src" / "farfan_core").exists() and (parent / "setup.py").exists():
886:             return parent
887:
888:     # Fallback: if we can't find it, assume we're in src/farfan_core/utils
889:     # and go up 3 levels
890:     return current.parent.parent.parent
891:
892:
893: # Global constants for common directories
```

```
894: PROJECT_ROOT: Final[Path] = _detect_project_root()
895: SRC_DIR: Final[Path] = PROJECT_ROOT / "src"
896: DATA_DIR: Final[Path] = PROJECT_ROOT / "data"
897: TESTS_DIR: Final[Path] = PROJECT_ROOT / "tests"
898:
899:
900: def proj_root() -> Path:
901:     """
902:     Get the project root directory.
903:
904:     Returns:
905:         Absolute path to the project root (where pyproject.toml lives)
906:     """
907:     return PROJECT_ROOT
908:
909:
910: def src_dir() -> Path:
911:     """Get the src directory path."""
912:     return SRC_DIR
913:
914:
915: def data_dir() -> Path:
916:     """
917:     Get the data directory path.
918:     Creates it if it doesn't exist.
919:     """
920:     DATA_DIR.mkdir(parents=True, exist_ok=True)
921:     return DATA_DIR
922:
923:
924: def tmp_dir() -> Path:
925:     """
926:     Get a project-specific temporary directory.
927:
928:     Uses PROJECT_ROOT/tmp to keep temporary files within the workspace
929:     and avoid polluting system temp directories.
930:
931:     Returns:
932:         Path to tmp directory (created if needed)
933:     """
934:     tmp = PROJECT_ROOT / "tmp"
935:     tmp.mkdir(parents=True, exist_ok=True)
936:     return tmp
937:
938:
939: def build_dir() -> Path:
940:     """
941:     Get the build directory for generated artifacts.
942:
943:     Returns:
944:         Path to build directory (created if needed)
945:     """
946:     build = PROJECT_ROOT / "build"
947:     build.mkdir(parents=True, exist_ok=True)
948:     return build
949:
```

```
def cache_dir() -> Path:
    """
    Get the cache directory.

    Returns:
        Path to cache directory (created if needed)
    """
    cache = build_dir() / "cache"
    cache.mkdir(parents=True, exist_ok=True)
    return cache


def reports_dir() -> Path:
    """
    Get the reports directory for generated reports.

    Returns:
        Path to reports directory (created if needed)
    """
    reports = build_dir() / "reports"
    reports.mkdir(parents=True, exist_ok=True)
    return reports


def is_within(base: Path, child: Path) -> bool:
    """
    Check if child path is within base directory (no traversal outside).

    Args:
        base: Base directory that should contain child
        child: Path to check

    Returns:
        True if child is within base, False otherwise

    Example:
        >>> project_root = Path("project_root")
        >>> is_within(project_root, project_root / "src" / "file.py")
        True
        >>> other_root = Path("other_project")
        >>> is_within(project_root, other_root / "file.py")
        False
    """
    try:
        base_resolved = base.resolve()
        child_resolved = child.resolve()

        # Check if child is relative to base
        child_resolved.relative_to(base_resolved)
        return True
    except (ValueError, RuntimeError):
        return False


def safe_join(base: Path, *parts: str) -> Path:
```

```
    """
    Safely join path components, preventing traversal outside base.

    This prevents directory traversal attacks using ".." components.

    Args:
        base: Base directory
        *parts: Path components to join

    Returns:
        Resolved path within base

    Raises:
        PathTraversalError: If the resulting path would be outside base

    Example:
        >>> project_root = Path("project_root")
        >>> safe_join(project_root, "src", "file.py")
        project_root/src/file.py
        >>> safe_join(project_root, "..", "other")  # raises
        PathTraversalError
    """
    result = base.joinpath(*parts).resolve()

    if not is_within(base, result):
        raise PathTraversalError(
            f"Path traversal detected: '{result}' is outside base '{base}'. "
            f"Use paths within the workspace."
        )

    return result


def normalize_unicode(path: Path, form: str = "NFC") -> Path:
    """
    Normalize Unicode in path for cross-platform consistency.

    Different filesystems handle Unicode differently:
    - macOS (HFS+) uses NFD normalization
    - Linux typically uses NFC
    - Windows uses UTF-16

    Args:
        path: Path to normalize
        form: Unicode normalization form ("NFC", "NFD", "NFKC", "NFKD")
              Default "NFC" for maximum compatibility

    Returns:
        Path with normalized Unicode
    """
    normalized_str = unicodedata.normalize(form, str(path))
    return Path(normalized_str)


def normalize_case(path: Path) -> Path:
    """
```

```
    Normalize path case for case-insensitive filesystems.

    On case-insensitive filesystems (Windows, macOS default), this ensures
    consistent casing. On case-sensitive systems (Linux), returns unchanged.

    Args:
        path: Path to normalize

    Returns:
        Path with normalized case
    """
    # Check if filesystem is case-sensitive
    # This is a heuristic - we check if we can create files differing only in case
    if path.exists():
        # Use actual case from filesystem
        try:
            # On Windows/macOS this will resolve to actual case
            return path.resolve()
        except Exception:
            pass

    return path


def resources(package: str, *path_parts: str) -> Path:
    """
    Access packaged resource files in a portable way.

    This uses importlib.resources (Python 3.9+) to access resources that
    are included in the installed package, whether from source or wheel.

    Args:
        package: Package name (e.g., "farfan_core.core")
        *path_parts: Path components within the package

    Returns:
        Path to the resource

    Raises:
        PathNotFoundError: If resource doesn't exist

    Example:
        >>> resources("farfan_core.core", "config", "default.yaml")
        Path('/path/to/farfan_core/core/config/default.yaml')
    """
    try:
        # Python 3.9+ way
        from importlib.resources import files

        pkg_path = files(package)
        for part in path_parts:
            pkg_path = pkg_path.joinpath(part)

        # Convert to Path - files() returns Traversable
        if hasattr(pkg_path, '__fspath__'):
            return Path(pkg_path)
```

```
        else:
            # Fallback for Traversable that doesn't support __fspath__
            # Read the resource and return a path to it
            raise PathNotFoundError(
                f"Resource '{'.'.join(path_parts)}' in package '{package}' "
                f"is not accessible as a filesystem path. "
                f"Consider using importlib.resources.read_text() or read_binary() instead."
            )
    except (ImportError, ModuleNotFoundError, FileNotFoundError, TypeError) as e:
        raise PathNotFoundError(
            f"Resource '{'/'.join(path_parts)}' not found in package '{package}'. "
            f"Ensure it's declared in pyproject.toml [tool.setuptools.package-data]. "
            f"Error: {e}"
        ) from e


def validate_read_path(path: Path) -> None:
    """
    Validate a path before reading from it.

    Args:
        path: Path to validate

    Raises:
        PathNotFoundError: If path doesn't exist
        PermissionError: If path is not readable
    """
    if not path.exists():
        raise PathNotFoundError(f"Path does not exist: '{path}'")

    if not os.access(path, os.R_OK):
        raise PermissionError(f"Path is not readable: '{path}'")


def validate_write_path(path: Path, allow_source_tree: bool = False) -> None:
    """
    Validate a path before writing to it.

    By default, prohibits writing to the source tree to prevent
    accidental modification of versioned code.

    Args:
        path: Path to validate
        allow_source_tree: If True, allow writing to source tree
                           (for special cases like code generation)

    Raises:
        PathOutsideWorkspaceError: If path is outside workspace
        PermissionError: If parent directory is not writable
        ValueError: If trying to write to source tree when not allowed
    """
    # Ensure it's within the workspace
    if not is_within(PROJECT_ROOT, path):
        raise PathOutsideWorkspaceError(
            f"Cannot write to '{path}' - outside workspace '{PROJECT_ROOT}'"
        )
```

```
1174:
1175:     # Prohibit writing to source tree unless explicitly allowed
1176:     if not allow_source_tree and is_within(SRC_DIR, path):
1177:         raise ValueError(
1178:             f"Cannot write to source tree: '{path}'. "
1179:             f"Write to build/, cache/, or reports/ instead. "
1180:             f"If you need to write to source (e.g., code generation), "
1181:             f"set allow_source_tree=True."
1182:         )
1183:
1184:     # Ensure parent directory exists and is writable
1185:     parent = path.parent
1186:     if parent.exists() and not os.access(parent, os.W_OK):
1187:         raise PermissionError(f"Parent directory is not writable: '{parent}'")
1188:
1189:
1190: # Environment variable accessors (typed and safe)
1191:
1192: def get_env_path(key: str, default: Path | None = None) -> Path | None:
1193:     """
1194:     Get a path from environment variable.
1195:
1196:     Args:
1197:         key: Environment variable name
1198:         default: Default value if not set
1199:
1200:     Returns:
1201:         Path from environment or default
1202:     """
1203:     value = os.getenv(key)
1204:     if value is None:
1205:         return default
1206:     return Path(value).resolve()
1207:
1208:
1209: def get_workdir() -> Path:
1210:     """
1211:     Get the working directory from FLUX_WORKDIR env var or default to project root.
1212:     """
1213:     return get_env_path("FLUX_WORKDIR", PROJECT_ROOT) or PROJECT_ROOT
1214:
1215:
1216: def get_tmpdir() -> Path:
1217:     """
1218:     Get the temporary directory from FLUX_TMPDIR env var or default to project tmp.
1219:     """
1220:     result = get_env_path("FLUX_TMPDIR", tmp_dir()) or tmp_dir()
1221:     result.mkdir(parents=True, exist_ok=True)
1222:     return result
1223:
1224:
1225: def get_reports_dir() -> Path:
1226:     """
1227:     Get the reports directory from FLUX_REPORTS env var or default to build/reports.
1228:     """
1229:     result = get_env_path("FLUX_REPORTS", reports_dir()) or reports_dir()
```

```
    result.mkdir(parents=True, exist_ok=True)
    return result


__all__ = [
    # Exceptions
    "PathError",
    "PathTraversalError",
    "PathNotFoundError",
    "PathOutsideWorkspaceError",
    "UnnormalizedPathError",
    # Constants
    "PROJECT_ROOT",
    "SRC_DIR",
    "DATA_DIR",
    "TESTS_DIR",
    # Directory accessors
    "proj_root",
    "src_dir",
    "data_dir",
    "tmp_dir",
    "build_dir",
    "cache_dir",
    "reports_dir",
    # Path operations
    "is_within",
    "safe_join",
    "normalize_unicode",
    "normalize_case",
    "resources",
    # Validation
    "validate_read_path",
    "validate_write_path",
    # Environment
    "get_env_path",
    "get_workdir",
    "get_tmpdir",
    "get_reports_dir",
]


================================================================================
FILE: src/farfan_pipeline/utils/proof_generator.py
================================================================================

"""Cryptographic proof generation for pipeline execution verification.

This module generates cryptographic proof files that allow non-engineers to verify
that a pipeline execution was successful and complete. It produces:
- proof.json: Contains execution metadata, phase/question counts, and SHA-256 hashes
- proof.hash: SHA-256 hash of the proof.json file for verification

The proof is ONLY generated when ALL success conditions are met:
- All phases report success=True
- No abort is active
```

```
- Non-empty artifacts exist (JSON/MD/logs)
"""

import hashlib
import json
from dataclasses import dataclass, field
from pathlib import Path
from typing import Any
from farfan_pipeline.core.calibration.decorators import calibrated_method


@dataclass
class ProofData:
    """Container for proof generation data.

    All fields must be populated from real execution data.
    No values should be invented or hardcoded.
    """
    run_id: str
    timestamp_utc: str
    phases_total: int
    phases_success: int
    questions_total: int
    questions_answered: int
    evidence_records: int
    monolith_hash: str
    questionnaire_hash: str
    catalog_hash: str
    method_map_hash: str
    code_signature: dict[str, str]

    # Optional fields for additional verification
    input_pdf_hash: str | None = None
    artifacts_manifest: dict[str, str] = field(default_factory=dict)
    execution_metadata: dict[str, Any] = field(default_factory=dict)

    # Calibration metadata for traceability
    calibration_version: str | None = None
    calibration_hash: str | None = None


def compute_file_hash(file_path: Path) -> str:
    """Compute SHA-256 hash of a file.

    Args:
        file_path: Path to the file to hash

    Returns:
        Hex string of SHA-256 hash

    Raises:
        FileNotFoundError: If file doesn't exist
    """
    if not file_path.exists():
        raise FileNotFoundError(f"Cannot hash missing file: {file_path}")

```

```
1342:     sha256 = hashlib.sha256()
1343:     with open(file_path, 'rb') as f:
1344:         # Read in chunks for large files
1345:         for chunk in iter(lambda: f.read(65536), b''):
1346:             sha256.update(chunk)
1347:     return sha256.hexdigest()
1348:
1349:
1350: def compute_dict_hash(data: dict[str, Any]) -> str:
1351:     """Compute SHA-256 hash of a dictionary.
1352:
1353:     The dictionary is serialized with sort_keys=True and ensure_ascii=True
1354:     to ensure deterministic hashing.
1355:
1356:     Args:
1357:         data: Dictionary to hash
1358:
1359:     Returns:
1360:         Hex string of SHA-256 hash
1361:     """
1362:     json_str = json.dumps(data, sort_keys=True, ensure_ascii=True, separators=(',', ':'))
1363:     return hashlib.sha256(json_str.encode('utf-8')).hexdigest()
1364:
1365:
1366: def compute_code_signatures(src_root: Path) -> dict[str, str]:
1367:     """Compute SHA-256 hashes of core orchestrator files.
1368:
1369:     Args:
1370:         src_root: Root path to the src/farfan_core directory
1371:
1372:     Returns:
1373:         Dictionary mapping filename to SHA-256 hash
1374:
1375:     Raises:
1376:         FileNotFoundError: If any required file is missing
1377:     """
1378:     core_files = {
1379:         'core.py': src_root / 'core' / 'orchestrator' / 'core.py',
1380:         'executors.py': src_root / 'core' / 'orchestrator' / 'executors.py',
1381:         'factory.py': src_root / 'core' / 'orchestrator' / 'factory.py',
1382:     }
1383:
1384:     signatures = {}
1385:     for name, path in core_files.items():
1386:         if not path.exists():
1387:             raise FileNotFoundError(f"Required core file missing: {path}")
1388:         signatures[name] = compute_file_hash(path)
1389:
1390:     return signatures
1391:
1392:
1393: def verify_success_conditions(
1394:     phase_results: list[Any],
1395:     abort_active: bool,
1396:     output_dir: Path,
1397: ) -> tuple[bool, list[str]]:
```

```
    """Verify that all success conditions are met before generating proof.

    Args:
        phase_results: List of PhaseResult objects from orchestrator
        abort_active: Whether an abort signal is active
        output_dir: Directory where artifacts should exist

    Returns:
        Tuple of (success: bool, errors: list[str])
    """
    errors = []

    # Check all phases succeeded
    if not phase_results:
        errors.append("No phase results available")
        return False, errors

    failed_phases = [
        i for i, result in enumerate(phase_results)
        if not result.success
    ]
    if failed_phases:
        errors.append(f"Phases failed: {failed_phases}")

    # Check no abort
    if abort_active:
        errors.append("Abort signal is active")

    # Check for artifacts (at minimum, directory should exist and have content)
    if not output_dir.exists():
        errors.append(f"Output directory does not exist: {output_dir}")
    else:
        # Check for at least some artifacts
        artifacts = list(output_dir.rglob('*.json')) + list(output_dir.rglob('*.md'))
        if not artifacts:
            errors.append(f"No artifacts (JSON/MD) found in {output_dir}")

    return len(errors) == 0, errors


def generate_proof(
    proof_data: ProofData,
    output_dir: Path,
) -> tuple[Path, Path]:
    """Generate proof.json and proof.hash files.

    Args:
        proof_data: Proof data to serialize
        output_dir: Directory where proof files will be written

    Returns:
        Tuple of (proof.json path, proof.hash path)

    Raises:
        ValueError: If proof_data is incomplete
    """
```

```
    # Validate required fields are not empty
    required_fields = [
        'run_id', 'timestamp_utc', 'monolith_hash', 'questionnaire_hash',
        'catalog_hash', 'code_signature'
    ]
    for field_name in required_fields:
        value = getattr(proof_data, field_name)
        if not value:
            raise ValueError(f"Required field '{field_name}' is empty")

    # Ensure output directory exists
    output_dir.mkdir(parents=True, exist_ok=True)

    # Build proof dictionary
    proof_dict = {
        'run_id': proof_data.run_id,
        'timestamp_utc': proof_data.timestamp_utc,
        'phases_total': proof_data.phases_total,
        'phases_success': proof_data.phases_success,
        'questions_total': proof_data.questions_total,
        'questions_answered': proof_data.questions_answered,
        'evidence_records': proof_data.evidence_records,
        'monolith_hash': proof_data.monolith_hash,
        'questionnaire_hash': proof_data.questionnaire_hash,
        'catalog_hash': proof_data.catalog_hash,
        'method_map_hash': proof_data.method_map_hash,
        'code_signature': proof_data.code_signature,
    }

    # Add optional fields if present
    if proof_data.input_pdf_hash:
        proof_dict['input_pdf_hash'] = proof_data.input_pdf_hash
    if proof_data.artifacts_manifest:
        proof_dict['artifacts_manifest'] = proof_data.artifacts_manifest
    if proof_data.execution_metadata:
        proof_dict['execution_metadata'] = proof_data.execution_metadata

    # Add calibration metadata for traceability
    if proof_data.calibration_version:
        proof_dict['calibration_version'] = proof_data.calibration_version
    if proof_data.calibration_hash:
        proof_dict['calibration_hash'] = proof_data.calibration_hash

    # Write proof.json with deterministic serialization
    proof_json_path = output_dir / 'proof.json'
    with open(proof_json_path, 'w', encoding='utf-8') as f:
        json.dump(
            proof_dict,
            f,
            sort_keys=True,
            ensure_ascii=True,
            separators=(',', ':'),
        )

    # Compute hash of proof.json (using compact serialization for hash)
    proof_hash = compute_dict_hash(proof_dict)
```

```
1510:
1511:     # Write proof.hash
1512:     proof_hash_path = output_dir / 'proof.hash'
1513:     with open(proof_hash_path, 'w', encoding='utf-8') as f:
1514:         f.write(proof_hash)
1515:
1516:     return proof_json_path, proof_hash_path
1517:
1518:
1519: def collect_artifacts_manifest(output_dir: Path) -> dict[str, str]:
1520:     """Collect hashes of all artifacts in output directory.
1521:
1522:     Args:
1523:         output_dir: Directory containing artifacts
1524:
1525:     Returns:
1526:         Dictionary mapping relative path to SHA-256 hash
1527:     """
1528:     manifest = {}
1529:
1530:     # Find all artifacts (JSON, MD, logs)
1531:     patterns = ['*.json', '*.md', '*.log', '*.txt']
1532:     for pattern in patterns:
1533:         for artifact_path in output_dir.rglob(pattern):
1534:             # Skip proof files themselves
1535:             if artifact_path.name in ('proof.json', 'proof.hash'):
1536:                 continue
1537:
1538:             try:
1539:                 rel_path = artifact_path.relative_to(output_dir)
1540:                 manifest[str(rel_path)] = compute_file_hash(artifact_path)
1541:             except (OSError, PermissionError, ValueError):
1542:                 # Skip files we can't read or hash
1543:                 pass
1544:
1545:     return manifest
1546:
1547:
1548: def verify_proof(proof_json_path: Path, proof_hash_path: Path) -> tuple[bool, str]:
1549:     """Verify that a proof.json file matches its proof.hash.
1550:
1551:     This allows anyone to verify that the proof hasn't been tampered with.
1552:
1553:     Args:
1554:         proof_json_path: Path to proof.json
1555:         proof_hash_path: Path to proof.hash
1556:
1557:     Returns:
1558:         Tuple of (valid: bool, message: str)
1559:     """
1560:     try:
1561:         # Read proof.json
1562:         with open(proof_json_path, encoding='utf-8') as f:
1563:             proof_dict = json.load(f)
1564:
1565:         # Read proof.hash
```

```
        with open(proof_hash_path, encoding='utf-8') as f:
            stored_hash = f.read().strip()

        # Recompute hash
        computed_hash = compute_dict_hash(proof_dict)

        # Compare
        if computed_hash == stored_hash:
            return True, "â\234\205 Proof verified: hash matches"
        else:
            return False, f"â\235\214 Proof verification failed: hash mismatch\n  Expected: {stored_hash}\n  Got: {computed_hash}"

    except Exception as e:
        return False, f"â\235\214 Proof verification error: {e}"


================================================================================
FILE: src/farfan_pipeline/utils/qmcm_hooks.py
================================================================================

"""
QMCM (Quality Method Call Monitoring) hooks for ReportAssemblyProducer

Records method calls for registry tracking and quality assurance.
Does NOT summarize or analyze - only records method invocations.
"""

import json
import logging
from datetime import datetime
from functools import wraps
from pathlib import Path
from typing import Any
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method

logger = logging.getLogger(__name__)

class QMCMRecorder:
    """
    Records method calls for quality monitoring

    Tracks:
    - Method invocations
    - Call frequency
    - Input/output types
    - Execution status

    Does NOT track:
    - Actual data content (no summarization leakage)
    - User-specific information
    """

    def __init__(self, recording_path: Path | None = None) -> None:
        """Initialize QMCM recorder"""
```

```
1622:         self.recording_path = recording_path or Path(".qmcm_recording.json")
1623:         self.calls: list[dict[str, Any]] = []
1624:         self.enabled = True
1625:
1626:     def record_call(
1627:             self,
1628:             method_name: str,
1629:             input_types: dict[str, str],
1630:             output_type: str,
1631:             execution_status: str = "success",
1632:             execution_time_ms: float = ParameterLoaderV2.get("farfan_core.utils.qmcm_hooks.QMCMRecorder.__init__", "auto_param_L45_39", 0.0),
1633:             monolith_hash: str | None = None
1634:     ) -> None:
1635:         """
1636:         Record a method call
1637:
1638:         Args:
1639:             method_name: Name of the method called
1640:             input_types: Dictionary mapping parameter names to type names
1641:             output_type: Type name of the return value
1642:             execution_status: 'success', 'error', or 'skipped'
1643:             execution_time_ms: Execution time in milliseconds
1644:             monolith_hash: SHA-256 hash of questionnaire_monolith.json (recommended)
1645:
1646:         ARCHITECTURAL NOTE: Including monolith_hash ties each method call
1647:         to the specific questionnaire version, enabling reproducibility.
1648:         Use factory.compute_monolith_hash() to generate this value.
1649:         """
1650:         if not self.enabled:
1651:             return
1652:
1653:         call_record = {
1654:             "timestamp": datetime.now().isoformat(),
1655:             "method_name": method_name,
1656:             "input_types": input_types,
1657:             "output_type": output_type,
1658:             "execution_status": execution_status,
1659:             "execution_time_ms": round(execution_time_ms, 2)
1660:         }
1661:
1662:         # Include monolith_hash if provided
1663:         if monolith_hash is not None:
1664:             call_record["monolith_hash"] = monolith_hash
1665:
1666:         self.calls.append(call_record)
1667:         logger.debug(f"QMCM recorded: {method_name}")
1668:
1669:     @calibrated_method("farfan_core.utils.qmcm_hooks.QMCMRecorder.get_statistics")
1670:     def get_statistics(self) -> dict[str, Any]:
1671:         """
1672:         Get recording statistics
1673:
1674:         Returns summary of method call patterns without data content
1675:         """
1676:         if not self.calls:
1677:             return {
```

```
                "total_calls": 0,
                "unique_methods": 0,
                "method_frequency": {},
                "success_rate": ParameterLoaderV2.get("farfan_core.utils.qmcm_hooks.QMCMRecorder.get_statistics", "auto_param_L94_32", 0.0),
                "most_called_method": None
            }

        method_counts = {}
        success_count = 0

        for call in self.calls:
            method_name = call["method_name"]
            method_counts[method_name] = method_counts.get(method_name, 0) + 1

            if call["execution_status"] == "success":
                success_count += 1

        most_called = None
        if method_counts:
            most_called = max(method_counts.items(), key=lambda x: x[1])[0]

        return {
            "total_calls": len(self.calls),
            "unique_methods": len(method_counts),
            "method_frequency": method_counts,
            "success_rate": success_count / len(self.calls) if self.calls else ParameterLoaderV2.get("farfan_core.utils.qmcm_hooks.QMCMRecorder.get_statistics", "auto_param_L116_79", 0.0),
            "most_called_method": most_called
        }

    @calibrated_method("farfan_core.utils.qmcm_hooks.QMCMRecorder.save_recording")
    def save_recording(self) -> None:
        """Save recording to disk"""
        recording_data = {
            "recording_metadata": {
                "generated_at": datetime.now().isoformat(),
                "total_calls": len(self.calls)
            },
            "statistics": self.get_statistics(),
            "calls": self.calls
        }

        with open(self.recording_path, 'w') as f:
            json.dump(recording_data, f, indent=2)

        logger.info(f"QMCM recording saved: {self.recording_path}")

    @calibrated_method("farfan_core.utils.qmcm_hooks.QMCMRecorder.load_recording")
    def load_recording(self) -> None:
        """Load recording from disk"""
        if not self.recording_path.exists():
            logger.warning(f"No recording found: {self.recording_path}")
            return

        with open(self.recording_path) as f:
            recording_data = json.load(f)
```

```
1733:
1734:         self.calls = recording_data.get("calls", [])
1735:         logger.info(f"QMCM recording loaded: {len(self.calls)} calls")
1736:
1737:     @calibrated_method("farfan_core.utils.qmcm_hooks.QMCMRecorder.clear_recording")
1738:     def clear_recording(self) -> None:
1739:         """Clear all recorded calls"""
1740:         self.calls = []
1741:         logger.info("QMCM recording cleared")
1742:
1743:     @calibrated_method("farfan_core.utils.qmcm_hooks.QMCMRecorder.enable")
1744:     def enable(self) -> None:
1745:         """Enable recording"""
1746:         self.enabled = True
1747:
1748:     @calibrated_method("farfan_core.utils.qmcm_hooks.QMCMRecorder.disable")
1749:     def disable(self) -> None:
1750:         """Disable recording"""
1751:         self.enabled = False
1752:
1753: # Global recorder instance
1754: _global_recorder: QMCMRecorder | None = None
1755:
1756: def get_global_recorder() -> QMCMRecorder:
1757:     """Get or create global QMCM recorder"""
1758:     global _global_recorder
1759:     if _global_recorder is None:
1760:         _global_recorder = QMCMRecorder()
1761:     return _global_recorder
1762:
1763: def qmcm_record(method=None, *, monolith_hash: str | None = None):
1764:     """
1765:     Decorator to record method calls in QMCM
1766:
1767:     Usage:
1768:         @qmcm_record
1769:         @calibrated_method("farfan_core.utils.qmcm_hooks.QMCMRecorder.my_method")
1770:         def my_method(self, arg1: str, arg2: int) -> dict:
1771:             return {"result": "data"}
1772:
1773:         # With monolith hash (recommended for questionnaire-dependent methods)
1774:         @qmcm_record(monolith_hash=compute_monolith_hash(questionnaire))
1775:         @calibrated_method("farfan_core.utils.qmcm_hooks.QMCMRecorder.my_questionnaire_method")
1776:         def my_questionnaire_method(self, question_id: str) -> dict:
1777:             return {"result": "data"}
1778:     """
1779:     def decorator(func):
1780:         @wraps(func)
1781:         def wrapper(*args, **kwargs):
1782:             recorder = get_global_recorder()
1783:
1784:             import time
1785:             start_time = time.time()
1786:
1787:             try:
1788:                 result = func(*args, **kwargs)
```

```
                execution_time_ms = (time.time() - start_time) * 1000

                # Record the call
                input_types = {}
                if args:
                    # Skip self argument
                    for i, arg in enumerate(args[1:], 1):
                        input_types[f"arg{i}"] = type(arg).__name__
                for key, value in kwargs.items():
                    input_types[key] = type(value).__name__

                output_type = type(result).__name__

                recorder.record_call(
                    method_name=func.__name__,
                    input_types=input_types,
                    output_type=output_type,
                    execution_status="success",
                    execution_time_ms=execution_time_ms,
                    monolith_hash=monolith_hash
                )

                return result

            except Exception:
                execution_time_ms = (time.time() - start_time) * 1000

                recorder.record_call(
                    method_name=func.__name__,
                    input_types={},
                    output_type="error",
                    execution_status="error",
                    execution_time_ms=execution_time_ms,
                    monolith_hash=monolith_hash
                )

                raise

        return wrapper

    # Handle both @qmcm_record and @qmcm_record() usage
    if method is None:
        return decorator
    else:
        return decorator(method)

# Export public API
__all__ = [
    'QMCMRecorder',
    'get_global_recorder',
    'qmcm_record'
]


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

```
FILE: src/farfan_pipeline/utils/runtime_error_fixes.py
================================================================================

"""
Runtime Error Fixes for Policy Analysis

This module contains fixes for three critical runtime errors:
1. 'bool' object is not iterable - Functions returning bool instead of list
2. 'str' object has no attribute 'text' - String passed where spacy object expected
3. can't multiply sequence by non-int of type 'float' - List multiplication by float

These fixes are applied defensively to prevent crashes in production.
"""

from typing import TYPE_CHECKING, Any
from farfan_pipeline.core.calibration.decorators import calibrated_method

if TYPE_CHECKING:
    import numpy as np
    NumpyArray = np.ndarray
else:
    NumpyArray = Any  # type: ignore[misc]

try:
    HAS_NUMPY = True
except ImportError:
    HAS_NUMPY = False

def ensure_list_return(value: Any) -> list[Any]:
    """
    Ensure a value is a list, converting bool/None to empty list.

    Fixes: 'bool' object is not iterable

    Args:
        value: Value that should be a list

    Returns:
        Empty list if value is False/None/bool, otherwise the value as-is
    """
    if isinstance(value, bool) or value is None:
        return []
    if isinstance(value, list):
        return value
    # If it's iterable but not a list, convert it
    try:
        return list(value)
    except (TypeError, ValueError):
        return []

def safe_text_extract(obj: Any) -> str:
    """
    Safely extract text from object that might be str or have .text attribute.

    Fixes: 'str' object has no attribute 'text'

```

```
    Args:
        obj: Object that is either str or has .text attribute (e.g., spacy Doc/Span)

    Returns:
        Extracted text string
    """
    # If it's already a string, return it
    if isinstance(obj, str):
        return obj

    # If it has a .text attribute, extract it
    if hasattr(obj, 'text'):
        text_value = obj.text
        if isinstance(text_value, str):
            return text_value

    # Fallback: convert to string
    return str(obj)

def safe_weighted_multiply(items: list[float] | NumpyArray, weight: float) -> list[float] | NumpyArray:
    """
    Safely multiply a list or array by a weight.

    Fixes: can't multiply sequence by non-int of type 'float'

    Args:
        items: List or array of numbers
        weight: Weight to multiply by

    Returns:
        New list/array with each element multiplied by weight
    """
    # If it's a numpy array, use numpy multiplication
    if HAS_NUMPY and hasattr(items, '__array_interface__'):
        import numpy as np  # Import here for runtime use
        if isinstance(items, np.ndarray):
            return items * weight

    # If it's a list, use list comprehension
    if isinstance(items, list):
        return [item * weight for item in items]

    # If it's something else iterable, convert and multiply
    try:
        return [item * weight for item in items]
    except (TypeError, ValueError):
        # If multiplication fails, return empty list
        return []

def safe_list_iteration(value: Any) -> list[Any]:
    """
    Ensure a value can be safely iterated over.

    Converts bool, None, or non-iterables to empty list.
    Handles the common error of trying to iterate over bool.

```

```
    Args:
        value: Value to iterate over

    Returns:
        Iterable list
    """
    # Reject booleans explicitly
    if isinstance(value, bool):
        return []

    # Handle None
    if value is None:
        return []

    # If it's already a list, return it
    if isinstance(value, list):
        return value

    # If it's a string, don't iterate over characters - return as single item
    if isinstance(value, str):
        return [value]

    # Try to convert to list
    try:
        return list(value)
    except (TypeError, ValueError):
        return []


================================================================================
FILE: src/farfan_pipeline/utils/schema_monitor.py
================================================================================

"""
SCHEMA DRIFT MONITORING - Watch Production Payloads
==================================================

Sample payloads in staging/prod and validate shapes.
Emit metrics on key presence/type.
Page when new keys appear or required keys vanish.

Catches upstream changes (or LLM output drift) instantly.
"""

from __future__ import annotations

import json
import logging
import random
from collections import Counter, defaultdict
from dataclasses import dataclass, field
from datetime import datetime
from typing import TYPE_CHECKING, Any, TypedDict
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method
```

```
2013:
2014: if TYPE_CHECKING:
2015:     from collections.abc import Mapping
2016:     from pathlib import Path
2017:
2018: logger = logging.getLogger(__name__)
2019:
2020: # ============================================================================
2021: # SCHEMA SHAPE TRACKING
2022: # ============================================================================
2023:
2024: class SchemaShape(TypedDict):
2025:     """Shape of a data payload."""
2026:
2027:     keys: set[str]
2028:     types: dict[str, str]
2029:     sample_values: dict[str, Any]
2030:     timestamp: str
2031:
2032: @dataclass
2033: class SchemaStats:
2034:     """Statistics about schema shape over time."""
2035:
2036:     key_frequency: Counter[str] = field(default_factory=Counter)
2037:     type_by_key: dict[str, Counter[str]] = field(default_factory=lambda: defaultdict(Counter))
2038:     new_keys: set[str] = field(default_factory=set)
2039:     missing_keys: set[str] = field(default_factory=set)
2040:     total_samples: int = 0
2041:     last_updated: datetime | None = None
2042:
2043: class SchemaDriftDetector:
2044:     """
2045:     Detects schema drift by sampling payloads and tracking shape changes.
2046:
2047:     Usage:
2048:         detector = SchemaDriftDetector(sample_rate=0.05)
2049:
2050:         # In your API/pipeline
2051:         if detector.should_sample():
2052:             detector.record_payload(data, source="api_input")
2053:
2054:         # Check for drift
2055:         alerts = detector.get_alerts()
2056:     """
2057:
2058:     def __init__(
2059:         self,
2060:         *,
2061:         sample_rate: float = 0.05,
2062:         baseline_path: Path | None = None,
2063:         alert_threshold: float = 0.1,
2064:     ) -> None:
2065:         """
2066:         Initialize schema drift detector.
2067:
2068:         Args:
```

```
2069:             sample_rate: Percentage of payloads to sample (0.01 = 1%, 0.05 = 5%)
2070:             baseline_path: Path to baseline schema file
2071:             alert_threshold: Threshold for drift alert (% of samples with drift)
2072:         """
2073:         self.sample_rate = sample_rate
2074:         self.baseline_path = baseline_path
2075:         self.alert_threshold = alert_threshold
2076:
2077:         # Tracking state
2078:         self.stats_by_source: dict[str, SchemaStats] = defaultdict(SchemaStats)
2079:         self.baseline_schema: dict[str, SchemaShape] = {}
2080:
2081:         # Load baseline if provided
2082:         if baseline_path and baseline_path.exists():
2083:             self._load_baseline()
2084:
2085:     @calibrated_method("farfan_core.utils.schema_monitor.SchemaDriftDetector.should_sample")
2086:     def should_sample(self) -> bool:
2087:         """Decide whether to sample this payload (probabilistic)."""
2088:         return random.random() < self.sample_rate
2089:
2090:     def record_payload(
2091:         self,
2092:         payload: Mapping[str, Any],
2093:         *,
2094:         source: str,
2095:         timestamp: datetime | None = None,
2096:     ) -> None:
2097:         """
2098:         Record a payload for schema tracking.
2099:
2100:         Args:
2101:             payload: Data payload to analyze
2102:             source: Source identifier (e.g., "api_input", "document_loader")
2103:             timestamp: Optional timestamp, defaults to now
2104:         """
2105:         if timestamp is None:
2106:             timestamp = datetime.utcnow()
2107:
2108:         stats = self.stats_by_source[source]
2109:
2110:         # Extract shape
2111:         keys = set(payload.keys())
2112:         types = {k: type(v).__name__ for k, v in payload.items()}
2113:
2114:         # Update statistics
2115:         stats.total_samples += 1
2116:         stats.last_updated = timestamp
2117:
2118:         for key in keys:
2119:             stats.key_frequency[key] += 1
2120:             stats.type_by_key[key][types[key]] += 1
2121:
2122:         # Detect new keys (compared to baseline)
2123:         if source in self.baseline_schema:
2124:             baseline_keys = self.baseline_schema[source]["keys"]
```

```
2125:             new_keys = keys - baseline_keys
2126:             if new_keys:
2127:                 stats.new_keys.update(new_keys)
2128:                 logger.warning(
2129:                     f"SCHEMA_DRIFT[source={source}]: New keys detected: {new_keys}"
2130:                 )
2131:
2132:             missing_keys = baseline_keys - keys
2133:             if missing_keys:
2134:                 stats.missing_keys.update(missing_keys)
2135:                 logger.warning(
2136:                     f"SCHEMA_DRIFT[source={source}]: Missing keys detected: {missing_keys}"
2137:                 )
2138:
2139:     @calibrated_method("farfan_core.utils.schema_monitor.SchemaDriftDetector.get_alerts")
2140:     def get_alerts(self, *, source: str | None = None) -> list[dict[str, Any]]:
2141:         """
2142:         Get schema drift alerts.
2143:
2144:         Args:
2145:             source: Optional source filter
2146:
2147:         Returns:
2148:             List of alert dicts
2149:         """
2150:         alerts: list[dict[str, Any]] = []
2151:
2152:         sources = [source] if source else list(self.stats_by_source.keys())
2153:
2154:         for src in sources:
2155:             stats = self.stats_by_source[src]
2156:
2157:             if stats.new_keys:
2158:                 alerts.append({
2159:                     "level": "WARNING",
2160:                     "source": src,
2161:                     "type": "NEW_KEYS",
2162:                     "keys": list(stats.new_keys),
2163:                     "timestamp": stats.last_updated.isoformat() if stats.last_updated else None,
2164:                 })
2165:
2166:             if stats.missing_keys:
2167:                 alerts.append({
2168:                     "level": "CRITICAL",
2169:                     "source": src,
2170:                     "type": "MISSING_KEYS",
2171:                     "keys": list(stats.missing_keys),
2172:                     "timestamp": stats.last_updated.isoformat() if stats.last_updated else None,
2173:                 })
2174:
2175:             # Check for type inconsistencies
2176:             for key, type_counts in stats.type_by_key.items():
2177:                 if len(type_counts) > 1:
2178:                     # Multiple types seen for same key
2179:                     dominant_type = type_counts.most_common(1)[0][0]
2180:                     other_types = [t for t in type_counts if t != dominant_type]
```

```
2181:
2182:                     alerts.append({
2183:                         "level": "WARNING",
2184:                         "source": src,
2185:                         "type": "TYPE_INCONSISTENCY",
2186:                         "key": key,
2187:                         "expected_type": dominant_type,
2188:                         "observed_types": other_types,
2189:                         "timestamp": stats.last_updated.isoformat() if stats.last_updated else None,
2190:                     })
2191:
2192:         return alerts
2193:
2194:     @calibrated_method("farfan_core.utils.schema_monitor.SchemaDriftDetector.save_baseline")
2195:     def save_baseline(self, output_path: Path) -> None:
2196:         """
2197:         Save current schema shapes as baseline.
2198:
2199:         Args:
2200:             output_path: Path to save baseline JSON
2201:         """
2202:         baseline: dict[str, dict[str, Any]] = {}
2203:
2204:         for source, stats in self.stats_by_source.items():
2205:             # Get most common keys (present in >50% of samples)
2206:             threshold = stats.total_samples * ParameterLoaderV2.get("farfan_core.utils.schema_monitor.SchemaDriftDetector.save_baseline", "auto_param_L215_4
6", 0.5)
2207:             common_keys = {
2208:                 key for key, count in stats.key_frequency.items()
2209:                 if count >= threshold
2210:             }
2211:
2212:             # Get dominant type for each key
2213:             types = {
2214:                 key: type_counts.most_common(1)[0][0]
2215:                 for key, type_counts in stats.type_by_key.items()
2216:             }
2217:
2218:             baseline[source] = {
2219:                 "keys": list(common_keys),
2220:                 "types": types,
2221:                 "timestamp": datetime.utcnow().isoformat(),
2222:             }
2223:
2224:         output_path.write_text(json.dumps(baseline, indent=2))
2225:         logger.info(f"Saved schema baseline to {output_path}")
2226:
2227:     @calibrated_method("farfan_core.utils.schema_monitor.SchemaDriftDetector._load_baseline")
2228:     def _load_baseline(self) -> None:
2229:         """Load baseline schema from file."""
2230:         if not self.baseline_path:
2231:             return
2232:
2233:         try:
2234:             data = json.loads(self.baseline_path.read_text())
2235:
```

```
            for source, shape_data in data.items():
                self.baseline_schema[source] = {
                    "keys": set(shape_data["keys"]),
                    "types": shape_data["types"],
                    "sample_values": {},
                    "timestamp": shape_data["timestamp"],
                }

            logger.info(f"Loaded schema baseline from {self.baseline_path}")
        except Exception as e:
            logger.error(f"Failed to load baseline: {e}")

    @calibrated_method("farfan_core.utils.schema_monitor.SchemaDriftDetector.get_metrics")
    def get_metrics(self, *, source: str | None = None) -> dict[str, Any]:
        """
        Get monitoring metrics.

        Args:
            source: Optional source filter

        Returns:
            Dict of metrics
        """
        if source:
            stats = self.stats_by_source.get(source)
            if not stats:
                return {}

            return {
                "source": source,
                "total_samples": stats.total_samples,
                "unique_keys": len(stats.key_frequency),
                "new_keys_count": len(stats.new_keys),
                "missing_keys_count": len(stats.missing_keys),
                "type_inconsistencies": sum(
                    1 for counts in stats.type_by_key.values()
                    if len(counts) > 1
                ),
            }

        # Aggregate across all sources
        return {
            "sources": list(self.stats_by_source.keys()),
            "total_samples": sum(s.total_samples for s in self.stats_by_source.values()),
            "sources_with_drift": len([
                s for s in self.stats_by_source.values()
                if s.new_keys or s.missing_keys
            ]),
        }

# ============================================================================
# PAYLOAD VALIDATOR
# ============================================================================

class PayloadValidator:
    """
```

```
    Validate payloads against expected schema.

    Usage:
        validator = PayloadValidator(schema_path=Path("schemas/api_input.json"))

        try:
            validator.validate(data, source="api_endpoint")
        except ValueError as e:
            logger.error(f"Validation failed: {e}")
    """

    def __init__(self, *, schema_path: Path | None = None) -> None:
        """
        Initialize payload validator.

        Args:
            schema_path: Path to schema definition JSON
        """
        self.schema_path = schema_path
        self.schemas: dict[str, dict[str, Any]] = {}

        if schema_path and schema_path.exists():
            self._load_schemas()

    def validate(
        self,
        payload: Mapping[str, Any],
        *,
        source: str,
        strict: bool = True,
    ) -> None:
        """
        Validate payload against schema.

        Args:
            payload: Data payload to validate
            source: Source identifier
            strict: If True, raise on missing keys; if False, only warn

        Raises:
            ValueError: If validation fails in strict mode
            TypeError: If value types don't match schema
        """
        if source not in self.schemas:
            logger.warning(f"No schema defined for source '{source}'")
            return

        schema = self.schemas[source]
        required_keys = set(schema.get("required_keys", []))
        expected_types = schema.get("types", {})

        # Check required keys
        payload_keys = set(payload.keys())
        missing = required_keys - payload_keys

        if missing:
```

```
2348:             msg = f"VALIDATION_ERROR[source={source}]: Missing required keys: {missing}"
2349:             if strict:
2350:                 raise ValueError(msg)
2351:             else:
2352:                 logger.warning(msg)
2353:
2354:         # Check types
2355:         for key, expected_type in expected_types.items():
2356:             if key in payload:
2357:                 actual_type = type(payload[key]).__name__
2358:                 if actual_type != expected_type:
2359:                     msg = (
2360:                         f"VALIDATION_ERROR[source={source}, key={key}]: "
2361:                         f"Expected type {expected_type}, got {actual_type}"
2362:                     )
2363:                     if strict:
2364:                         raise TypeError(msg)
2365:                     else:
2366:                         logger.warning(msg)
2367:
2368:     @calibrated_method("farfan_core.utils.schema_monitor.PayloadValidator._load_schemas")
2369:     def _load_schemas(self) -> None:
2370:         """Load schema definitions from file."""
2371:         if not self.schema_path:
2372:             return
2373:
2374:         try:
2375:             self.schemas = json.loads(self.schema_path.read_text())
2376:             logger.info(f"Loaded schemas from {self.schema_path}")
2377:         except Exception as e:
2378:             logger.error(f"Failed to load schemas: {e}")
2379:
2380: # ============================================================================
2381: # GLOBAL INSTANCE (optional convenience)
2382: # ============================================================================
2383:
2384: # Singleton detector for application-wide use
2385: _global_detector: SchemaDriftDetector | None = None
2386:
2387: def get_detector() -> SchemaDriftDetector:
2388:     """Get or create global schema drift detector."""
2389:     global _global_detector
2390:     if _global_detector is None:
2391:         _global_detector = SchemaDriftDetector(sample_rate=ParameterLoaderV2.get("farfan_core.utils.schema_monitor.PayloadValidator._load_schemas", "auto_pa
ram_L400_59", 0.05))
2392:     return _global_detector
2393:
2394:
2395:
2396: ================================================================================
2397: FILE: src/farfan_pipeline/utils/seed_factory.py
2398: ================================================================================
2399:
2400: """
2401: Deterministic Seed Factory
2402: Generates reproducible seeds for all stochastic operations
```

```
2403: """
2404:
2405: import hashlib
2406: import hmac
2407: import random
2408: from typing import Any
2409: from farfan_pipeline.core.calibration.decorators import calibrated_method
2410:
2411: try:
2412:     import numpy as np
2413:     NUMPY_AVAILABLE = True
2414: except ImportError:
2415:     NUMPY_AVAILABLE = False
2416:     np = None  # type: ignore
2417:
2418: class SeedFactory:
2419:     """
2420:     Factory for generating deterministic seeds
2421:
2422:     Ensures:
2423:     - Reproducibility: Same inputs â\206\222 same seed
2424:     - Uniqueness: Different contexts â\206\222 different seeds
2425:     - Cryptographic quality: HMAC-SHA256 derivation
2426:     """
2427:
2428:     # Fixed salt for seed derivation (should be configured per deployment)
2429:     DEFAULT_SALT = b"PDM_EVALUATOR_V2_DETERMINISTIC_SEED_2025"
2430:
2431:     def __init__(self, fixed_salt: bytes | None = None) -> None:
2432:         self.salt = fixed_salt or self.DEFAULT_SALT
2433:
2434:     def create_deterministic_seed(
2435:         self,
2436:         correlation_id: str,
2437:         file_checksums: dict[str, str] | None = None,
2438:         context: dict[str, Any] | None = None
2439:     ) -> int:
2440:         """
2441:         Generate deterministic seed from correlation ID and context
2442:
2443:         Args:
2444:             correlation_id: Unique workflow instance identifier
2445:             file_checksums: Dict of {filename: sha256_checksum}
2446:             context: Additional context (question_id, policy_area, etc.)
2447:
2448:         Returns:
2449:             32-bit integer seed (0 to 2^32-1)
2450:
2451:         Example:
2452:             >>> factory = SeedFactory()
2453:             >>> seed1 = factory.create_deterministic_seed("run-001", {"data.json": "abc123"})
2454:             >>> seed2 = factory.create_deterministic_seed("run-001", {"data.json": "abc123"})
2455:             >>> assert seed1 == seed2  # Reproducible
2456:         """
2457:
2458:         # Build deterministic input string
```

```
        components = [correlation_id]

        # Add file checksums (sorted for determinism)
        if file_checksums:
            sorted_checksums = sorted(file_checksums.items())
            for filename, checksum in sorted_checksums:
                components.append(f"{filename}:{checksum}")

        # Add context (sorted for determinism)
        if context:
            sorted_context = sorted(context.items())
            for key, value in sorted_context:
                components.append(f"{key}={value}")

        # Combine with deterministic separator
        seed_input = "|".join(components).encode('utf-8')

        # HMAC-SHA256 for cryptographic quality
        seed_hmac = hmac.new(
            key=self.salt,
            msg=seed_input,
            digestmod=hashlib.sha256
        )

        # Convert to 32-bit integer
        seed_bytes = seed_hmac.digest()[:4]  # First 4 bytes
        seed_int = int.from_bytes(seed_bytes, byteorder='big')

        return seed_int

    @calibrated_method("farfan_core.utils.seed_factory.SeedFactory.configure_global_random_state")
    def configure_global_random_state(self, seed: int) -> None:
        """
        Configure all random number generators with seed

        Sets:
        - Python random module
        - NumPy random state
        - (Add torch, tensorflow if needed)

        Args:
            seed: Deterministic seed
        """

        # Python random module
        random.seed(seed)

        # NumPy
        if NUMPY_AVAILABLE and np is not None:
            np.random.seed(seed)

        # TODO: Add torch.manual_seed(seed) if PyTorch is used
        # TODO: Add tf.random.set_seed(seed) if TensorFlow is used

class DeterministicContext:
    """
```

```
    Context manager for deterministic execution

    Usage:
        with DeterministicContext(correlation_id="run-001") as seed:
            # All random operations are deterministic
            result = some_stochastic_function()
    """

    def __init__(
        self,
        correlation_id: str,
        file_checksums: dict[str, str] | None = None,
        context: dict[str, Any] | None = None,
        fixed_salt: bytes | None = None
    ) -> None:
        self.correlation_id = correlation_id
        self.file_checksums = file_checksums
        self.context = context
        self.factory = SeedFactory(fixed_salt=fixed_salt)

        self.seed: int | None = None
        self.previous_random_state = None
        self.previous_numpy_state = None

    def __enter__(self) -> int:
        """Enter deterministic context"""

        # Generate deterministic seed
        self.seed = self.factory.create_deterministic_seed(
            correlation_id=self.correlation_id,
            file_checksums=self.file_checksums,
            context=self.context
        )

        # Save current random states
        self.previous_random_state = random.getstate()
        if NUMPY_AVAILABLE and np is not None:
            self.previous_numpy_state = np.random.get_state()

        # Configure with deterministic seed
        self.factory.configure_global_random_state(self.seed)

        return self.seed

    def __exit__(self, exc_type, exc_val, exc_tb):
        """Exit deterministic context and restore previous state"""

        # Restore previous random states
        if self.previous_random_state:
            random.setstate(self.previous_random_state)

        if NUMPY_AVAILABLE and np is not None and self.previous_numpy_state:
            np.random.set_state(self.previous_numpy_state)

        return False

```

```
def create_deterministic_seed(
    correlation_id: str,
    file_checksums: dict[str, str] | None = None,
    **context_kwargs
) -> int:
    """
    Convenience function for creating deterministic seed

    Args:
        correlation_id: Unique workflow instance ID
        file_checksums: Dict of file checksums
        **context_kwargs: Additional context as keyword arguments

    Returns:
        Deterministic 32-bit integer seed

    Example:
        >>> seed = create_deterministic_seed(
        ...     "run-001",
        ...     question_id="P1-D1-Q001",
        ...     policy_area="P1"
        ... )
    """
    factory = SeedFactory()
    return factory.create_deterministic_seed(
        correlation_id=correlation_id,
        file_checksums=file_checksums,
        context=context_kwargs if context_kwargs else None
    )


================================================================================
FILE: src/farfan_pipeline/utils/signature_validator.py
================================================================================

"""
Signature Validation and Interface Governance System
=====================================================

Implements automated signature consistency auditing, runtime validation,
and interface governance to prevent function signature mismatches.

Based on the Strategic Mitigation Plan for addressing interface inconsistencies.

Author: Signature Governance Team
Version: 1.0.0
"""

import ast
import functools
import hashlib
import inspect
import json
import logging
from collections.abc import Callable
```

```
2627: from dataclasses import asdict, dataclass, field
2628: from datetime import datetime
2629: from pathlib import Path
2630: from typing import Any, TypeVar, get_type_hints
2631: from farfan_pipeline.core.calibration.decorators import calibrated_method
2632:
2633: logger = logging.getLogger(__name__)
2634:
2635: # Type variable for decorated functions
2636: F = TypeVar('F', bound=Callable[..., Any])
2637:
2638: # ============================================================================
2639: # SIGNATURE METADATA STORAGE
2640: # ============================================================================
2641:
2642: @dataclass
2643: class FunctionSignature:
2644:     """Stores metadata about a function's signature"""
2645:     module: str
2646:     class_name: str | None
2647:     function_name: str
2648:     parameters: list[str]
2649:     parameter_types: dict[str, str]
2650:     return_type: str
2651:     signature_hash: str
2652:     timestamp: str = field(default_factory=lambda: datetime.now().isoformat())
2653:
2654:     @calibrated_method("farfan_core.utils.signature_validator.FunctionSignature.to_dict")
2655:     def to_dict(self) -> dict[str, Any]:
2656:         return asdict(self)
2657:
2658: class SignatureRegistry:
2659:     """
2660:     Maintains a registry of function signatures with version tracking
2661:     Implements signature snapshotting as described in the mitigation plan
2662:     """
2663:
2664:     def __init__(self, registry_path: Path = Path("data/signature_registry.json")) -> None:
2665:         self.registry_path = registry_path
2666:         self.signatures: dict[str, FunctionSignature] = {}
2667:         self.load()
2668:
2669:     @calibrated_method("farfan_core.utils.signature_validator.SignatureRegistry.compute_signature_hash")
2670:     def compute_signature_hash(self, func: Callable) -> str:
2671:         """Compute a hash of the function's signature"""
2672:         sig = inspect.signature(func)
2673:         sig_str = str(sig)
2674:         return hashlib.sha256(sig_str.encode()).hexdigest()[:16]
2675:
2676:     @calibrated_method("farfan_core.utils.signature_validator.SignatureRegistry.register_function")
2677:     def register_function(self, func: Callable) -> FunctionSignature:
2678:         """Register a function's signature"""
2679:         sig = inspect.signature(func)
2680:
2681:         # Extract parameter information
2682:         parameters = list(sig.parameters.keys())
```

```
2683:
2684:         # Get type hints if available
2685:         try:
2686:             type_hints = get_type_hints(func)
2687:             parameter_types = {
2688:                 name: str(type_hints.get(name, 'Any'))
2689:                 for name in parameters
2690:             }
2691:             return_type = str(type_hints.get('return', 'Any'))
2692:         except (TypeError, AttributeError, NameError) as e:
2693:             # get_type_hints can fail for various reasons:
2694:             # - TypeError: if func is not a callable
2695:             # - AttributeError: if func doesn't have required attributes
2696:             # - NameError: if type hints reference undefined names
2697:             logger.debug(f"Could not extract type hints for {func.__name__}: {e}")
2698:             parameter_types = dict.fromkeys(parameters, 'Any')
2699:             return_type = 'Any'
2700:
2701:         # Get module and class information
2702:         module = func.__module__ if hasattr(func, '__module__') else 'unknown'
2703:         class_name = None
2704:         if hasattr(func, '__qualname__') and '.' in func.__qualname__:
2705:             class_name = func.__qualname__.rsplit('.', 1)[0]
2706:
2707:         signature_hash = self.compute_signature_hash(func)
2708:
2709:         func_sig = FunctionSignature(
2710:             module=module,
2711:             class_name=class_name,
2712:             function_name=func.__name__,
2713:             parameters=parameters,
2714:             parameter_types=parameter_types,
2715:             return_type=return_type,
2716:             signature_hash=signature_hash
2717:         )
2718:
2719:         # Store in registry
2720:         key = self._get_function_key(module, class_name, func.__name__)
2721:         self.signatures[key] = func_sig
2722:
2723:         return func_sig
2724:
2725:     @calibrated_method("farfan_core.utils.signature_validator.SignatureRegistry._get_function_key")
2726:     def _get_function_key(self, module: str, class_name: str | None, func_name: str) -> str:
2727:         """Generate a unique key for a function"""
2728:         if class_name:
2729:             return f"{module}.{class_name}.{func_name}"
2730:         return f"{module}.{func_name}"
2731:
2732:     @calibrated_method("farfan_core.utils.signature_validator.SignatureRegistry.get_signature")
2733:     def get_signature(self, module: str, class_name: str | None, func_name: str) -> FunctionSignature | None:
2734:         """Retrieve a stored signature"""
2735:         key = self._get_function_key(module, class_name, func_name)
2736:         return self.signatures.get(key)
2737:
2738:     @calibrated_method("farfan_core.utils.signature_validator.SignatureRegistry.has_signature_changed")
```

```
    def has_signature_changed(self, func: Callable) -> tuple[bool, FunctionSignature | None, FunctionSignature | None]:
        """Check if a function's signature has changed from the registered version"""
        module = func.__module__ if hasattr(func, '__module__') else 'unknown'
        class_name = None
        if hasattr(func, '__qualname__') and '.' in func.__qualname__:
            class_name = func.__qualname__.rsplit('.', 1)[0]

        old_sig = self.get_signature(module, class_name, func.__name__)
        if old_sig is None:
            return False, None, None  # No previous signature

        new_sig = self.register_function(func)
        changed = old_sig.signature_hash != new_sig.signature_hash

        return changed, old_sig, new_sig

    @calibrated_method("farfan_core.utils.signature_validator.SignatureRegistry.save")
    def save(self) -> None:
        """Save registry to disk"""
        self.registry_path.parent.mkdir(parents=True, exist_ok=True)

        registry_data = {
            key: sig.to_dict()
            for key, sig in self.signatures.items()
        }

        with open(self.registry_path, 'w') as f:
            json.dump(registry_data, f, indent=2)

        logger.info(f"Saved {len(self.signatures)} signatures to {self.registry_path}")

    @calibrated_method("farfan_core.utils.signature_validator.SignatureRegistry.load")
    def load(self) -> None:
        """Load registry from disk"""
        if not self.registry_path.exists():
            logger.info(f"No existing registry found at {self.registry_path}")
            return

        try:
            with open(self.registry_path) as f:
                registry_data = json.load(f)

            self.signatures = {
                key: FunctionSignature(**data)
                for key, data in registry_data.items()
            }

            logger.info(f"Loaded {len(self.signatures)} signatures from {self.registry_path}")
        except Exception as e:
            logger.error(f"Failed to load registry: {e}")

# Global registry instance
_signature_registry = SignatureRegistry()

# ============================================================================
# RUNTIME VALIDATION DECORATOR
```

```
2795: # ============================================================================
2796:
2797: def validate_signature(enforce: bool = True, track: bool = True):
2798:     """
2799:     Decorator to validate function calls against expected signatures at runtime
2800:
2801:     Args:
2802:         enforce: If True, raise TypeError on signature violations
2803:         track: If True, register signature in the global registry
2804:
2805:     Example:
2806:         @validate_signature(enforce=True)
2807:         def my_function(arg1: str, arg2: int) -> bool:
2808:             return True
2809:     """
2810:     def decorator(func: F) -> F:
2811:         # Register function signature if tracking is enabled
2812:         if track:
2813:             _signature_registry.register_function(func)
2814:
2815:         # Get the function signature
2816:         sig = inspect.signature(func)
2817:
2818:         @functools.wraps(func)
2819:         def wrapper(*args, **kwargs):
2820:             # Bind arguments to signature
2821:             try:
2822:                 bound_args = sig.bind(*args, **kwargs)
2823:                 bound_args.apply_defaults()
2824:             except TypeError as e:
2825:                 error_msg = (
2826:                     f"Signature mismatch in {func.__module__}.{func.__qualname__}: {e}\n"
2827:                     f"Expected signature: {sig}\n"
2828:                     f"Called with args: {args}, kwargs: {kwargs}"
2829:                 )
2830:                 logger.error(error_msg)
2831:
2832:                 if enforce:
2833:                     raise TypeError(error_msg) from e
2834:                 else:
2835:                     logger.warning(f"Signature validation failed but enforcement is disabled: {e}")
2836:
2837:             # Call the original function
2838:             return func(*args, **kwargs)
2839:
2840:         return wrapper  # type: ignore
2841:
2842:     return decorator
2843:
2844: def validate_call_signature(func: Callable, *args, **kwargs) -> bool:
2845:     """
2846:     Validate that a function call matches the expected signature without actually calling it
2847:
2848:     Args:
2849:         func: Function to validate
2850:         *args: Positional arguments
```

```
2851:         **kwargs: Keyword arguments
2852:
2853:     Returns:
2854:         True if signature is valid, False otherwise
2855:     """
2856:     try:
2857:         sig = inspect.signature(func)
2858:         sig.bind(*args, **kwargs)
2859:         return True
2860:     except TypeError:
2861:         return False
2862:
2863: # ============================================================================
2864: # STATIC SIGNATURE AUDITOR
2865: # ============================================================================
2866:
2867: @dataclass
2868: class SignatureMismatch:
2869:     """Represents a detected signature mismatch"""
2870:     caller_module: str
2871:     caller_function: str
2872:     caller_line: int
2873:     callee_module: str
2874:     callee_class: str | None
2875:     callee_function: str
2876:     expected_signature: str
2877:     actual_call: str
2878:     severity: str  # 'high', 'medium', 'low'
2879:     description: str
2880:
2881: class SignatureAuditor:
2882:     """
2883:     Static introspection tool to cross-validate function definitions against call sites
2884:     Implements automated signature consistency audit from the mitigation plan
2885:     """
2886:
2887:     def __init__(self) -> None:
2888:         self.mismatches: list[SignatureMismatch] = []
2889:         self.call_graph: dict[str, list[tuple[str, int, list[str], dict[str, str]]]] = {}
2890:
2891:     @calibrated_method("farfan_core.utils.signature_validator.SignatureAuditor.audit_module")
2892:     def audit_module(self, module_path: Path) -> list[SignatureMismatch]:
2893:         """
2894:         Audit a Python module for signature mismatches
2895:
2896:         Args:
2897:             module_path: Path to the Python module
2898:
2899:         Returns:
2900:             List of detected signature mismatches
2901:         """
2902:         logger.info(f"Auditing module: {module_path}")
2903:
2904:         # Skip test files, virtual environments, and build directories
2905:         exclude_patterns = ['test', 'venv', '.venv', '__pycache__', '.git', 'build', 'dist']
2906:         if any(module_path.match(f'*/{pattern}/*') or module_path.match(f'*/{pattern}')
```

```
2907:                for pattern in exclude_patterns):
2908:            logger.debug(f"Skipping excluded path: {module_path}")
2909:            return []
2910:
2911:        try:
2912:            with open(module_path, encoding='utf-8') as f:
2913:                source_code = f.read()
2914:
2915:            tree = ast.parse(source_code, filename=str(module_path))
2916:
2917:            # Extract function definitions
2918:            function_defs = self._extract_function_definitions(tree, module_path.stem)
2919:
2920:            # Extract function calls
2921:            function_calls = self._extract_function_calls(tree, module_path.stem)
2922:
2923:            # Cross-validate
2924:            mismatches = self._cross_validate(function_defs, function_calls)
2925:
2926:            self.mismatches.extend(mismatches)
2927:
2928:            return mismatches
2929:
2930:        except Exception as e:
2931:            logger.error(f"Failed to audit {module_path}: {e}")
2932:            return []
2933:
2934:    @calibrated_method("farfan_core.utils.signature_validator.SignatureAuditor._extract_function_definitions")
2935:    def _extract_function_definitions(self, tree: ast.AST, module_name: str) -> dict[str, ast.FunctionDef]:
2936:        """Extract all function definitions from AST"""
2937:        functions = {}
2938:
2939:        for node in ast.walk(tree):
2940:            if isinstance(node, ast.FunctionDef):
2941:                # Generate full qualified name
2942:                full_name = f"{module_name}.{node.name}"
2943:                functions[full_name] = node
2944:
2945:        return functions
2946:
2947:    @calibrated_method("farfan_core.utils.signature_validator.SignatureAuditor._extract_function_calls")
2948:    def _extract_function_calls(self, tree: ast.AST, module_name: str) -> list[tuple[str, int, ast.Call]]:
2949:        """Extract all function calls from AST"""
2950:        calls = []
2951:
2952:        for node in ast.walk(tree):
2953:            if isinstance(node, ast.Call):
2954:                # Try to get the function name
2955:                func_name = None
2956:                if isinstance(node.func, ast.Name):
2957:                    func_name = node.func.id
2958:                elif isinstance(node.func, ast.Attribute):
2959:                    func_name = node.func.attr
2960:
2961:                if func_name:
2962:                    calls.append((func_name, node.lineno, node))
```

```
2963:
2964:         return calls
2965:
2966:     def _cross_validate(
2967:         self,
2968:         function_defs: dict[str, ast.FunctionDef],
2969:         function_calls: list[tuple[str, int, ast.Call]]
2970:     ) -> list[SignatureMismatch]:
2971:         """Cross-validate function calls against definitions"""
2972:         mismatches = []
2973:
2974:         # This is a simplified implementation
2975:         # A full implementation would need more sophisticated analysis
2976:
2977:         return mismatches
2978:
2979:     @calibrated_method("farfan_core.utils.signature_validator.SignatureAuditor.export_report")
2980:     def export_report(self, output_path: Path) -> None:
2981:         """Export audit report to JSON"""
2982:         output_path.parent.mkdir(parents=True, exist_ok=True)
2983:
2984:         report = {
2985:             "audit_timestamp": datetime.now().isoformat(),
2986:             "total_mismatches": len(self.mismatches),
2987:             "mismatches": [asdict(m) for m in self.mismatches]
2988:         }
2989:
2990:         with open(output_path, 'w') as f:
2991:             json.dump(report, f, indent=2)
2992:
2993:         logger.info(f"Exported audit report to {output_path}")
2994:
2995: # ============================================================================
2996: # COMPATIBILITY LAYER
2997: # ============================================================================
2998:
2999: def create_adapter(
3000:     func: Callable,
3001:     old_params: list[str],
3002:     new_params: list[str],
3003:     param_mapping: dict[str, str] | None = None
3004: ) -> Callable:
3005:     """
3006:     Create a backward-compatible adapter for a function with changed signature
3007:
3008:     Args:
3009:         func: The new function with updated signature
3010:         old_params: List of old parameter names
3011:         new_params: List of new parameter names
3012:         param_mapping: Optional mapping from old to new parameter names
3013:
3014:     Returns:
3015:         Adapter function that accepts old signature and calls new function
3016:     """
3017:     param_mapping = param_mapping or {}
3018:
```

```
3019:     @functools.wraps(func)
3020:     def adapter(*args, **kwargs):
3021:         # Remap old parameter names to new ones
3022:         new_kwargs = {}
3023:         for old_key, value in kwargs.items():
3024:             new_key = param_mapping.get(old_key, old_key)
3025:             new_kwargs[new_key] = value
3026:
3027:         return func(*args, **new_kwargs)
3028:
3029:     return adapter
3030:
3031: # ==============================================================================
3032: # MODULE INITIALIZATION
3033: # ==============================================================================
3034:
3035: def initialize_signature_registry(project_root: Path) -> None:
3036:     """
3037:     Initialize signature registry by scanning all Python files in the project
3038:
3039:     Args:
3040:         project_root: Root directory of the project
3041:     """
3042:     logger.info(f"Initializing signature registry for project: {project_root}")
3043:
3044:     python_files = list(project_root.glob("**/*.py"))
3045:     logger.info(f"Found {len(python_files)} Python files")
3046:
3047:     # This would require dynamic import which is complex
3048:     # For now, we rely on decorators to register functions
3049:
3050:     _signature_registry.save()
3051:
3052: def audit_project_signatures(project_root: Path, output_path: Path | None = None) -> list[SignatureMismatch]:
3053:     """
3054:     Audit all Python files in a project for signature mismatches
3055:
3056:     Args:
3057:         project_root: Root directory of the project
3058:         output_path: Optional path to save audit report
3059:
3060:     Returns:
3061:         List of detected signature mismatches
3062:     """
3063:     auditor = SignatureAuditor()
3064:
3065:     python_files = list(project_root.glob("**/*.py"))
3066:     logger.info(f"Auditing {len(python_files)} Python files")
3067:
3068:     # Define patterns to exclude
3069:     exclude_patterns = ['test', 'venv', '.venv', '__pycache__', '.git', 'build', 'dist']
3070:
3071:     all_mismatches = []
3072:     for py_file in python_files:
3073:         # Skip excluded patterns
3074:         if any(py_file.match(f'*/{pattern}/*') or py_file.match(f'*/{pattern}')
```

```
                for pattern in exclude_patterns):
            continue

        mismatches = auditor.audit_module(py_file)
        all_mismatches.extend(mismatches)

    if output_path:
        auditor.export_report(output_path)

    logger.info(f"Audit complete: {len(all_mismatches)} mismatches detected")

    return all_mismatches

# ============================================================================
# CLI INTERFACE
# ============================================================================

# Note: Main entry point removed to maintain I/O boundary separation.
# For CLI usage, see examples/ directory or create a dedicated CLI script.



================================================================================
FILE: src/farfan_pipeline/utils/spc_adapter.py
================================================================================

"""SPC to Orchestrator Adapter (Shim).

This module is a shim for backward compatibility. The canonical implementation
has been moved to `farfan_core.utils.cpp_adapter` to align with the Canon Policy Package (CPP)
terminology.

Please use `farfan_core.utils.cpp_adapter.CPPAdapter` instead.
"""

from __future__ import annotations

import warnings
from farfan_pipeline.utils.cpp_adapter import (
    CPPAdapter as SPCAdapter,
    CPPAdapterError as SPCAdapterError,
    adapt_cpp_to_orchestrator as adapt_spc_to_orchestrator
)
from farfan_pipeline.core.calibration.decorators import calibrated_method

# Issue deprecation warning when module is imported
warnings.warn(
    "farfan_core.utils.spc_adapter is deprecated. Use farfan_core.utils.cpp_adapter instead.",
    DeprecationWarning,
    stacklevel=2
)

__all__ = [
    'SPCAdapter',
    'SPCAdapterError',
    'adapt_spc_to_orchestrator',
```

```
]


================================================================================
FILE: src/farfan_pipeline/utils/validation/__init__.py
================================================================================

"""Validation module for pre-execution checks and preconditions."""

from farfan_pipeline.utils.validation.aggregation_models import (
    AggregationWeights,
    AreaAggregationConfig,
    ClusterAggregationConfig,
    DimensionAggregationConfig,
    MacroAggregationConfig,
    validate_dimension_config,
    validate_weights,
)
from farfan_pipeline.utils.validation.architecture_validator import (
    ArchitectureValidationResult,
    validate_architecture,
    write_validation_report,
)
from farfan_pipeline.utils.validation.golden_rule import GoldenRuleValidator, GoldenRuleViolation
from farfan_pipeline.utils.validation.schema_validator import (
    MonolithIntegrityReport,
    MonolithSchemaValidator,
    SchemaInitializationError,
    validate_monolith_schema,
)

__all__ = [
    "ArchitectureValidationResult",
    "GoldenRuleValidator",
    "GoldenRuleViolation",
    "validate_architecture",
    "write_validation_report",
    "AggregationWeights",
    "DimensionAggregationConfig",
    "AreaAggregationConfig",
    "ClusterAggregationConfig",
    "MacroAggregationConfig",
    "validate_weights",
    "validate_dimension_config",
    "MonolithSchemaValidator",
    "MonolithIntegrityReport",
    "SchemaInitializationError",
    "validate_monolith_schema",
]


================================================================================
FILE: src/farfan_pipeline/utils/validation/aggregation_models.py
================================================================================
```

```
3187:
3188: """
3189: Pydantic models for aggregation weight validation.
3190:
3191: This module provides strict type-safe validation for aggregation weights,
3192: ensuring zero-tolerance for invalid values at ingestion time.
3193: """
3194:
3195: from pydantic import BaseModel, ConfigDict, Field, field_validator, model_validator
3196: from typing_extensions import Self
3197: from farfan_pipeline.core.parameters import ParameterLoaderV2
3198: from farfan_pipeline.core.calibration.decorators import calibrated_method
3199:
3200:
3201: class AggregationWeights(BaseModel):
3202:     """
3203:     Validation model for aggregation weights.
3204:
3205:     Enforces:
3206:     - All weights must be non-negative (>= 0)
3207:     - All weights must be <= 1.0
3208:     - Weights must sum to 1.0 (within tolerance)
3209:     """
3210:
3211:     model_config = ConfigDict(frozen=True, extra='forbid')
3212:
3213:     weights: list[float] = Field(..., min_length=1, description="List of aggregation weights")
3214:     tolerance: float = Field(default=1e-6, ge=0, description="Tolerance for sum validation")
3215:
3216:     @field_validator('weights')
3217:     @classmethod
3218:     def validate_non_negative(cls, v: list[float]) -> list[float]:
3219:         """Ensure all weights are non-negative."""
3220:         for i, weight in enumerate(v):
3221:             if weight < 0:
3222:                 raise ValueError(
3223:                     f"Invalid aggregation weight at index {i}: {weight}. "
3224:                     f"All weights must be non-negative (>= 0)."
3225:                 )
3226:             if weight > 1.0:
3227:                 raise ValueError(
3228:                     f"Invalid aggregation weight at index {i}: {weight}. "
3229:                     f"All weights must be <= 1.0."
3230:                 )
3231:         return v
3232:
3233:     @model_validator(mode='after')
3234:     @calibrated_method("farfan_core.utils.validation.aggregation_models.AggregationWeights.validate_sum")
3235:     def validate_sum(self) -> Self:
3236:         """Ensure weights sum to ParameterLoaderV2.get("farfan_core.utils.validation.aggregation_models.AggregationWeights.validate_sum", "auto_param_L48_33
", 1.0) within tolerance."""
3237:         weight_sum = sum(self.weights)
3238:         expected_sum = ParameterLoaderV2.get("farfan_core.utils.validation.aggregation_models.AggregationWeights.validate_sum", "auto_param_L52_79", 1.0)
3239:         diff = abs(weight_sum - expected_sum)
3240:         if diff > self.tolerance:
3241:             raise ValueError(
```

```
                f"Weight sum validation failed: sum={weight_sum:.6f}, expected={expected_sum}. "
                f"Difference {diff:.6f} exceeds tolerance {self.tolerance:.6f}."
            )
        return self

class DimensionAggregationConfig(BaseModel):
    """Configuration for dimension-level aggregation."""

    model_config = ConfigDict(frozen=True, extra='forbid')

    dimension_id: str = Field(..., pattern=r'^DIM\d{2}$')
    area_id: str = Field(..., pattern=r'^PA\d{2}$')
    weights: AggregationWeights | None = None
    expected_question_count: int = Field(default=5, ge=1, le=10)
    group_by_keys: list[str] = Field(default=['dimension', 'policy_area'], min_length=1)


class AreaAggregationConfig(BaseModel):
    """Configuration for area-level aggregation."""

    model_config = ConfigDict(frozen=True, extra='forbid')

    area_id: str = Field(..., pattern=r'^PA\d{2}$')
    expected_dimension_count: int = Field(default=6, ge=1, le=10)
    weights: AggregationWeights | None = None
    group_by_keys: list[str] = Field(default=['area_id'], min_length=1)


class ClusterAggregationConfig(BaseModel):
    """Configuration for cluster-level aggregation."""

    model_config = ConfigDict(frozen=True, extra='forbid')

    cluster_id: str = Field(..., pattern=r'^CL\d{2}$')
    policy_area_ids: list[str] = Field(..., min_length=1)
    weights: AggregationWeights | None = None
    group_by_keys: list[str] = Field(default=['cluster_id'], min_length=1)

    @field_validator('policy_area_ids')
    @classmethod
    def validate_policy_areas(cls, v: list[str]) -> list[str]:
        """Ensure all policy area IDs follow the correct pattern."""
        for pa_id in v:
            if len(pa_id) < 3 or not pa_id.startswith('PA') or not pa_id[2:].isdigit():
                raise ValueError(f"Invalid policy area ID: {pa_id}. Expected format: PA##")
        return v

class MacroAggregationConfig(BaseModel):
    """Configuration for macro-level aggregation."""

    model_config = ConfigDict(frozen=True, extra='forbid')

    cluster_ids: list[str] = Field(..., min_length=1)
    weights: AggregationWeights | None = None

    @field_validator('cluster_ids')
```

```
    @classmethod
    def validate_clusters(cls, v: list[str]) -> list[str]:
        """Ensure all cluster IDs follow the correct pattern."""
        for cl_id in v:
            if len(cl_id) < 3 or not cl_id.startswith('CL') or not cl_id[2:].isdigit():
                raise ValueError(f"Invalid cluster ID: {cl_id}. Expected format: CL##")
        return v

def validate_weights(weights: list[float], tolerance: float = 1e-6) -> AggregationWeights:
    """
    Convenience function to validate a list of weights.

    Args:
        weights: List of weights to validate
        tolerance: Tolerance for sum validation

    Returns:
        Validated AggregationWeights instance

    Raises:
        ValueError: If validation fails
    """
    return AggregationWeights(weights=weights, tolerance=tolerance)

def validate_dimension_config(
    dimension_id: str,
    area_id: str,
    weights: list[float] | None = None,
    expected_question_count: int = 5
) -> DimensionAggregationConfig:
    """
    Validate dimension aggregation configuration.

    Args:
        dimension_id: Dimension ID (e.g., "DIM01")
        area_id: Area ID (e.g., "PA01")
        weights: Optional list of weights
        expected_question_count: Expected number of questions

    Returns:
        Validated configuration

    Raises:
        ValueError: If validation fails
    """
    weight_model = None
    if weights is not None:
        weight_model = validate_weights(weights)

    return DimensionAggregationConfig(
        dimension_id=dimension_id,
        area_id=area_id,
        weights=weight_model,
        expected_question_count=expected_question_count
    )

```

```
3354:
3355:
3356: ================================================================================
3357: FILE: src/farfan_pipeline/utils/validation/architecture_validator.py
3358: ================================================================================
3359:
3360: """Architecture validation utilities for the municipal policy analysis system.
3361:
3362: This module provides helpers to enforce that the municipal policy
3363: analysis architecture blueprint references real, implemented methods.
3364: It parses the ``policy_analysis_architecture.json`` specification,
3365: compares every referenced method against the inventoried codebase and
3366: produces coverage reports per analytical dimension.
3367: """
3368:
3369: from __future__ import annotations
3370:
3371: import ast
3372: import json
3373: import re
3374: from collections.abc import Iterable, Mapping
3375: from dataclasses import dataclass, field
3376: from pathlib import Path
3377: from farfan_pipeline.core.calibration.decorators import calibrated_method
3378:
3379: # Regular expression used to capture fully-qualified method references such as
3380: # ``ClassName.method_name``.
3381: METHOD_PATTERN = re.compile(r"^[A-Za-z_][A-Za-z0-9_]*\.[A-Za-z_][A-Za-z0-9_]*$")
3382:
3383: _EXTERNAL_REFERENCE = object()
3384:
3385: ALIAS_MAP: dict[str, object] = {
3386:     # Performance analyzer exposes the functionality through a private helper.
3387:     "PerformanceAnalyzer.analyze_loss_function": "PerformanceAnalyzer._calculate_loss_functions",
3388:     # The municipal plan analyzer generates recommendations with a private helper.
3389:     "PDETMunicipalPlanAnalyzer.generate_recommendations": "PDETMunicipalPlanAnalyzer._generate_recommendations",
3390:     # Advanced DAG validation leverages TeoriaCambio utilities internally.
3391:     "AdvancedDAGValidator.validacion_completa": "TeoriaCambio.validacion_completa",
3392:     "AdvancedDAGValidator._validar_orden_causal": "TeoriaCambio._validar_orden_causal",
3393:     "AdvancedDAGValidator._encontrar_caminos_completos": "TeoriaCambio._encontrar_caminos_completos",
3394:     # External dependency references (networkx graphs).
3395:     "nx.DiGraph": _EXTERNAL_REFERENCE,
3396: }
3397:
3398: # Root directory of the repository (two levels above this file).
3399: ROOT_DIR = Path(__file__).resolve().parent.parent
3400:
3401: @dataclass(frozen=True)
3402: class ArchitectureValidationResult:
3403:     """Container with the outcome of the architecture validation process."""
3404:
3405:     resolved_methods: set[str]
3406:     missing_methods: Mapping[str, Mapping[str, list[str]]]
3407:     coverage: float
3408:     total_spec_methods: int
3409:     total_available_methods: int
```

```
    per_dimension: Mapping[str, Mapping[str, list[str]]]
    global_methods: tuple[str, ...] = field(default_factory=tuple)

    @calibrated_method("farfan_core.utils.validation.architecture_validator.ArchitectureValidationResult.to_dict")
    def to_dict(self) -> dict[str, object]:
        """Serialise the validation result into a JSON-compatible dict."""

        return {
            "coverage": self.coverage,
            "total_spec_methods": self.total_spec_methods,
            "total_available_methods": self.total_available_methods,
            "resolved_methods": sorted(self.resolved_methods),
            "missing_methods": {
                dimension: dict(question_map.items())
                for dimension, question_map in self.missing_methods.items()
            },
            "per_dimension": {
                dimension: dict(question_map.items())
                for dimension, question_map in self.per_dimension.items()
            },
            "global_methods": list(self.global_methods),
        }

def load_architecture_spec(path: Path) -> dict[str, object]:
    """Load the JSON architecture specification from ``path``."""

    with path.open("r", encoding="utf-8") as handle:
        return json.load(handle)

def _extract_method_from_entry(entry: object) -> str | None:
    """Return the method string encoded in ``entry`` if present."""

    if isinstance(entry, str) and METHOD_PATTERN.match(entry):
        return entry

    if isinstance(entry, Mapping):
        # Architecture steps are stored as {"Class.method": "description"}
        for key in entry:
            if isinstance(key, str) and METHOD_PATTERN.match(key):
                return key

        # Some entries are dictionaries using {"name": "method"}. These lack
        # class information and therefore cannot be enforced reliably.
        name = entry.get("name") if isinstance(entry.get("name"), str) else None
        if name and METHOD_PATTERN.match(name):
            return name

    return None

def _extract_methods_from_string(value: str) -> Iterable[str]:
    """Extract additional method references embedded in textual descriptions."""

    for candidate in re.findall(r"[A-Za-z_][A-Za-z0-9_]*\.[A-Za-z_][A-Za-z0-9_]*", value):
        if METHOD_PATTERN.match(candidate):
            yield candidate

```

```
3466: def extract_architecture_methods(spec: Mapping[str, object]) -> tuple[dict[str, dict[str, list[str]]], list[str]]:
3467:     """Extract method sequences per dimension and global method references."""
3468:
3469:     policy_spec = spec.get("policy_analysis_architecture", {})
3470:     if not isinstance(policy_spec, Mapping):
3471:         raise ValueError("Malformed architecture specification: missing 'policy_analysis_architecture'.")
3472:
3473:     per_dimension: dict[str, dict[str, list[str]]] = {}
3474:     global_methods: list[str] = []
3475:
3476:     # --- Component level methods ---------------------------------------------------
3477:     orchestration = policy_spec.get("orchestration_flow", {})
3478:     if isinstance(orchestration, Mapping):
3479:         for component in orchestration.get("components", []):
3480:             if not isinstance(component, Mapping):
3481:                 continue
3482:             for entry in component.get("key_methods", []):
3483:                 method = _extract_method_from_entry(entry)
3484:                 if method:
3485:                     global_methods.append(method.strip())
3486:                 if isinstance(entry, Mapping):
3487:                     description = entry.get("description")
3488:                     if isinstance(description, str):
3489:                         global_methods.extend(list(_extract_methods_from_string(description)))
3490:
3491:     # --- Phase 0 (initialisation) --------------------------------------------------
3492:     phase_zero = policy_spec.get("phase_0_inicializacion_y_carga", {})
3493:     if isinstance(phase_zero, Mapping):
3494:         for step in phase_zero.get("steps", []):
3495:             if not isinstance(step, Mapping):
3496:                 continue
3497:             for action in step.get("actions", []):
3498:                 method = _extract_method_from_entry(action)
3499:                 if method:
3500:                     global_methods.append(method.strip())
3501:                 if isinstance(action, Mapping):
3502:                     for value in action.values():
3503:                         if isinstance(value, str):
3504:                             global_methods.extend(list(_extract_methods_from_string(value)))
3505:
3506:     # --- Analytical dimensions -----------------------------------------------------
3507:     for dimension in policy_spec.get("dimensiones", []):
3508:         if not isinstance(dimension, Mapping):
3509:             continue
3510:         dim_id = str(dimension.get("id", "UNKNOWN"))
3511:         dimension_methods: dict[str, list[str]] = {}
3512:         for subdimension in dimension.get("subdimension", []):
3513:             if not isinstance(subdimension, Mapping):
3514:                 continue
3515:             question_id = str(subdimension.get("pregunta", "UNKNOWN"))
3516:             methods: list[str] = []
3517:             for step in subdimension.get("cadena_metodos", []):
3518:                 method = _extract_method_from_entry(step)
3519:                 if method:
3520:                     methods.append(method.strip())
3521:                 if isinstance(step, Mapping):
```

```
                    for value in step.values():
                        if isinstance(value, str):
                            methods.extend(list(_extract_methods_from_string(value)))
            dimension_methods[question_id] = methods
        if dimension_methods:
            per_dimension[dim_id] = dimension_methods

    # --- Transversal modules -----------------------------------------------------
    transversal = policy_spec.get("modulos_transversales", {})
    if isinstance(transversal, Mapping):
        metricas = transversal.get("metricas_rendimiento", {})
        if isinstance(metricas, Mapping):
            for component in metricas.get("componentes", []):
                if not isinstance(component, Mapping):
                    continue
                method = _extract_method_from_entry(component)
                if method:
                    global_methods.append(method.strip())
                description = component.get("descripcion") or component.get("description")
                if isinstance(description, str):
                    global_methods.extend(list(_extract_methods_from_string(description)))

    return per_dimension, global_methods

def load_method_inventory(path: Path) -> tuple[set[str], set[str]]:
    """Load available class methods and module functions from the inventory."""

    with path.open("r", encoding="utf-8") as handle:
        inventory = json.load(handle)

    available_methods: set[str] = set()
    functions: set[str] = set()

    candidate_files = inventory.get("files", {})
    for file_name in candidate_files:
        file_path = ROOT_DIR / file_name
        if not file_path.exists():
            continue
        try:
            tree = ast.parse(file_path.read_text(encoding="utf-8"))
        except SyntaxError:
            continue

        for node in tree.body:
            if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef)):
                functions.add(node.name)
            elif isinstance(node, ast.ClassDef):
                for item in node.body:
                    if isinstance(item, (ast.FunctionDef, ast.AsyncFunctionDef)):
                        available_methods.add(f"{node.name}.{item.name}")

    return available_methods, functions

def _resolve_method_reference(
    reference: str,
    available_methods: set[str],
```

```
    available_functions: set[str],
) -> str | None:
    """Resolve a method reference using the available inventory."""

    reference = reference.strip()

    alias_target = ALIAS_MAP.get(reference)
    if alias_target is _EXTERNAL_REFERENCE:
        return reference
    if isinstance(alias_target, str):
        if alias_target == reference:
            return reference if reference in available_methods else None
        resolved_alias = _resolve_method_reference(alias_target, available_methods, available_functions)
        if resolved_alias:
            return resolved_alias

    if METHOD_PATTERN.match(reference):
        if reference in available_methods:
            return reference
        # Allow ``Class.init`` aliases for ``Class.__init__``
        if reference.endswith(".init"):
            init_alias = reference[:-4] + "__init__"
            if init_alias in available_methods:
                return init_alias
        return None

    # Plain function reference
    if reference in available_functions:
        return reference
    return None

def validate_architecture(spec_path: Path, inventory_path: Path) -> ArchitectureValidationResult:
    """Validate that every method described in the architecture exists."""

    spec = load_architecture_spec(spec_path)
    per_dimension, global_methods = extract_architecture_methods(spec)

    available_methods, available_functions = load_method_inventory(inventory_path)

    resolved_methods: set[str] = set()
    missing_methods: dict[str, dict[str, list[str]]] = {}

    # Validate global references
    for method in global_methods:
        resolved = _resolve_method_reference(method, available_methods, available_functions)
        if resolved:
            resolved_methods.add(resolved)
        else:
            missing_methods.setdefault("__global__", {}).setdefault("__global__", []).append(method)

    # Validate per-dimension references
    for dimension, question_map in per_dimension.items():
        for question, methods in question_map.items():
            for method in methods:
                resolved = _resolve_method_reference(method, available_methods, available_functions)
                if resolved:
```

```
                    resolved_methods.add(resolved)
                else:
                    missing_methods.setdefault(dimension, {}).setdefault(question, []).append(method)

    total_references = len(global_methods)
    total_references += sum(len(methods) for question_map in per_dimension.values() for methods in question_map.values())
    total_references = max(total_references, 1)

    coverage = len(resolved_methods) / total_references

    return ArchitectureValidationResult(
        resolved_methods=resolved_methods,
        missing_methods=missing_methods,
        coverage=coverage,
        total_spec_methods=total_references,
        total_available_methods=len(available_methods),
        per_dimension=per_dimension,
        global_methods=tuple(global_methods),
    )

def write_validation_report(result: ArchitectureValidationResult, output_path: Path) -> None:
    """Write the validation report to ``output_path`` in JSON format."""

    output_path.parent.mkdir(parents=True, exist_ok=True)
    with output_path.open("w", encoding="utf-8") as handle:
        json.dump(result.to_dict(), handle, indent=2, ensure_ascii=False)

def main() -> None:
    """Entry point for CLI usage."""

    spec_path = ROOT_DIR / "policy_analysis_architecture.json"
    inventory_path = ROOT_DIR / "COMPLETE_METHOD_CLASS_MAP.json"
    report_path = ROOT_DIR / "validation" / "architecture_validation_report.json"

    result = validate_architecture(spec_path, inventory_path)
    write_validation_report(result, report_path)

    if result.missing_methods:
        missing_total = sum(len(methods) for dimension in result.missing_methods.values() for methods in dimension.values())
        print(
            f"Architecture validation completed with {missing_total} missing methods. "
            f"Report saved to {report_path}."
        )
    else:
        print(f"Architecture validation successful. Report saved to {report_path}.")

# Note: Main entry point removed to maintain I/O boundary separation.
# For usage, call main() function from a script or see examples/ directory.


================================================================================
FILE: src/farfan_pipeline/utils/validation/contract_logger.py
================================================================================

#!/usr/bin/env python3
```

```
"""
Contract Error Logger - Structured logging for contract validation errors.

Provides a unified interface for logging contract violations in a machine-readable
format conforming to schemas/contract_error_log.schema.json.

Usage:
    from farfan_pipeline.utils.validation.contract_logger import ContractErrorLogger

    logger = ContractErrorLogger(module_name="scoring")
    logger.log_contract_mismatch(
        function="apply_scoring",
        key="confidence",
        needed="float",
        got=evidence.get("confidence"),
        file=__file__,
        line=234,
        remediation="Convert confidence to float between 0.0 and 1.0"
    )
"""

import json
import sys
import traceback
import uuid
from datetime import datetime, timezone
from typing import Any
from farfan_pipeline.core.calibration.decorators import calibrated_method


class ContractErrorLogger:
    """Structured logger for contract validation errors."""

    # Standard error codes
    ERR_CONTRACT_MISMATCH = "ERR_CONTRACT_MISMATCH"
    ERR_TYPE_VIOLATION = "ERR_TYPE_VIOLATION"
    ERR_SCHEMA_VALIDATION = "ERR_SCHEMA_VALIDATION"
    ERR_MISSING_REQUIRED_FIELD = "ERR_MISSING_REQUIRED_FIELD"
    ERR_INVALID_MODALITY = "ERR_INVALID_MODALITY"
    ERR_DETERMINISM_VIOLATION = "ERR_DETERMINISM_VIOLATION"

    # Severity levels
    CRITICAL = "CRITICAL"
    ERROR = "ERROR"
    WARNING = "WARNING"
    INFO = "INFO"

    def __init__(self, module_name: str, enable_stdout: bool = True) -> None:
        """
        Initialize contract error logger.

        Args:
            module_name: Name of the module using this logger
            enable_stdout: Whether to output to stdout (default: True)
        """
        self.module_name = module_name
```

```
3746:         self.enable_stdout = enable_stdout
3747:         self.request_id = str(uuid.uuid4())
3748:
3749:     def _log(
3750:         self,
3751:         error_code: str,
3752:         function: str,
3753:         message: str,
3754:         severity: str,
3755:         context: dict,
3756:         remediation: str | None = None,
3757:         stack_trace: list[str] | None = None
3758:     ) -> None:
3759:         """
3760:         Internal method to log structured error.
3761:
3762:         Args:
3763:             error_code: Standard error code
3764:             function: Function name where error occurred
3765:             message: Human-readable error message
3766:             severity: Error severity level
3767:             context: Structured context dictionary
3768:             remediation: Optional remediation steps
3769:             stack_trace: Optional stack trace
3770:         """
3771:         log_entry = {
3772:             "error_code": error_code,
3773:             "timestamp": datetime.now(timezone.utc).isoformat(),
3774:             "severity": severity,
3775:             "function": f"{self.module_name}.{function}",
3776:             "message": message,
3777:             "context": context,
3778:             "request_id": self.request_id
3779:         }
3780:
3781:         if remediation:
3782:             log_entry["remediation"] = remediation
3783:
3784:         if stack_trace:
3785:             log_entry["stack_trace"] = stack_trace
3786:
3787:         # Output as single-line JSON
3788:         log_line = json.dumps(log_entry, separators=(',', ':'))
3789:
3790:         if self.enable_stdout:
3791:             print(log_line, file=sys.stderr)
3792:
3793:     def log_contract_mismatch(
3794:         self,
3795:         function: str,
3796:         key: str,
3797:         needed: Any,
3798:         got: Any,
3799:         index: int | None = None,
3800:         file: str | None = None,
3801:         line: int | None = None,
```

```
3802:         remediation: str | None = None
3803:     ) -> None:
3804:         """
3805:         Log a contract mismatch error (ERR_CONTRACT_MISMATCH).
3806:
3807:         Args:
3808:             function: Function name where error occurred
3809:             key: Parameter/field name that failed
3810:             needed: Expected type/value
3811:             got: Actual value received
3812:             index: Optional index in collection
3813:             file: Optional source file
3814:             line: Optional line number
3815:             remediation: Optional remediation steps
3816:         """
3817:         context = {
3818:             "key": key,
3819:             "needed": needed,
3820:             "got": got
3821:         }
3822:
3823:         if index is not None:
3824:             context["index"] = index
3825:         if file:
3826:             context["file"] = file
3827:         if line:
3828:             context["line"] = line
3829:
3830:         message = f"Contract violation: required parameter '{key}' is missing or invalid"
3831:         if got is None:
3832:             message = f"Contract violation: required parameter '{key}' is missing"
3833:
3834:         self._log(
3835:             error_code=self.ERR_CONTRACT_MISMATCH,
3836:             function=function,
3837:             message=message,
3838:             severity=self.ERROR,
3839:             context=context,
3840:             remediation=remediation
3841:         )
3842:
3843:     def log_type_violation(
3844:         self,
3845:         function: str,
3846:         key: str,
3847:         expected_type: str,
3848:         got: Any,
3849:         file: str | None = None,
3850:         line: int | None = None,
3851:         remediation: str | None = None
3852:     ) -> None:
3853:         """
3854:         Log a type violation error (ERR_TYPE_VIOLATION).
3855:
3856:         Args:
3857:             function: Function name where error occurred
```

```
            key: Parameter/field name with wrong type
            expected_type: Expected type name
            got: Actual value received
            file: Optional source file
            line: Optional line number
            remediation: Optional remediation steps
        """
        actual_type = type(got).__name__

        context = {
            "key": key,
            "needed": expected_type,
            "got": str(got) if got is not None else None
        }

        if file:
            context["file"] = file
        if line:
            context["line"] = line

        message = f"Type violation: expected {expected_type} for '{key}', got {actual_type}"

        self._log(
            error_code=self.ERR_TYPE_VIOLATION,
            function=function,
            message=message,
            severity=self.ERROR,
            context=context,
            remediation=remediation
        )

    def log_invalid_modality(
        self,
        function: str,
        modality: str,
        allowed_modalities: list[str],
        file: str | None = None,
        line: int | None = None
    ) -> None:
        """
        Log an invalid modality error (ERR_INVALID_MODALITY).

        Args:
            function: Function name where error occurred
            modality: Invalid modality value
            allowed_modalities: List of allowed modalities
            file: Optional source file
            line: Optional line number
        """
        context = {
            "key": "modality",
            "needed": "|".join(allowed_modalities),
            "got": modality
        }

        if file:
```

```
3914:             context["file"] = file
3915:         if line:
3916:             context["line"] = line
3917:
3918:         message = f"Invalid modality: {modality} is not in allowed modalities"
3919:         remediation = f"Use one of the allowed modality types: {', '.join(allowed_modalities)}"
3920:
3921:         self._log(
3922:             error_code=self.ERR_INVALID_MODALITY,
3923:             function=function,
3924:             message=message,
3925:             severity=self.ERROR,
3926:             context=context,
3927:             remediation=remediation
3928:         )
3929:
3930:     def log_determinism_violation(
3931:         self,
3932:         function: str,
3933:         description: str,
3934:         expected_hash: str,
3935:         actual_hash: str,
3936:         file: str | None = None,
3937:         line: int | None = None
3938:     ) -> None:
3939:         """
3940:         Log a determinism violation (ERR_DETERMINISM_VIOLATION).
3941:
3942:         Args:
3943:             function: Function name where error occurred
3944:             description: Description of what failed determinism check
3945:             expected_hash: Expected hash value
3946:             actual_hash: Actual hash value
3947:             file: Optional source file
3948:             line: Optional line number
3949:         """
3950:         context = {
3951:             "key": "determinism_check",
3952:             "needed": expected_hash,
3953:             "got": actual_hash
3954:         }
3955:
3956:         if file:
3957:             context["file"] = file
3958:         if line:
3959:             context["line"] = line
3960:
3961:         message = f"Determinism violation: {description}"
3962:         remediation = "Check for non-deterministic operations (random, time, concurrency)"
3963:
3964:         # Include stack trace for determinism violations
3965:         stack_trace = traceback.format_stack()
3966:
3967:         self._log(
3968:             error_code=self.ERR_DETERMINISM_VIOLATION,
3969:             function=function,
```

```
3970:             message=message,
3971:             severity=self.CRITICAL,
3972:             context=context,
3973:             remediation=remediation,
3974:             stack_trace=stack_trace
3975:         )
3976:
3977: # Example usage
3978: # Note: Example usage removed to maintain I/O boundary separation.
3979: # For usage examples, see examples/ directory.
3980:
3981:
3982:
3983: ================================================================================
3984: FILE: src/farfan_pipeline/utils/validation/golden_rule.py
3985: ================================================================================
3986:
3987: """Golden Rule enforcement utilities."""
3988:
3989: from __future__ import annotations
3990:
3991: import hashlib
3992: from typing import TYPE_CHECKING
3993: from farfan_pipeline.core.calibration.decorators import calibrated_method
3994:
3995: if TYPE_CHECKING:
3996:     from collections.abc import Iterable
3997:
3998: class GoldenRuleViolation(Exception):
3999:     """Raised when a Golden Rule assertion is violated."""
4000:
4001: class GoldenRuleValidator:
4002:     """Enforces the Golden Rules across orchestrated execution phases."""
4003:
4004:     def __init__(self, questionnaire_hash: str, step_catalog: Iterable[str]) -> None:
4005:         self._baseline_questionnaire_hash = questionnaire_hash
4006:         self._baseline_step_signature = self._hash_sequence(step_catalog)
4007:         self._baseline_step_catalog = list(step_catalog)
4008:         self._state_ids: set[int] = set()
4009:         self._predicate_signature: set[str] | None = None
4010:
4011:     @staticmethod
4012:     def _hash_sequence(sequence: Iterable[str]) -> str:
4013:         canonical = "|".join(str(item) for item in sequence)
4014:         return hashlib.sha256(canonical.encode("utf-8")).hexdigest()
4015:
4016:     def assert_immutable_metadata(
4017:         self,
4018:         questionnaire_hash: str,
4019:         step_catalog: Iterable[str]
4020:     ) -> None:
4021:         """Ensure canonical questionnaire and step catalog remain unchanged."""
4022:
4023:         if self._baseline_questionnaire_hash:
4024:             if questionnaire_hash and questionnaire_hash != self._baseline_questionnaire_hash:
4025:                 raise GoldenRuleViolation("Questionnaire metadata hash mismatch")
```

```
4026:
4027:         if self._hash_sequence(step_catalog) != self._baseline_step_signature:
4028:             raise GoldenRuleViolation("Execution step catalog mutated")
4029:
4030:     @calibrated_method("farfan_core.utils.validation.golden_rule.GoldenRuleValidator.reset_atomic_state")
4031:     def reset_atomic_state(self) -> None:
4032:         """Reset atomic state tracking between phases."""
4033:
4034:         self._state_ids.clear()
4035:
4036:     @calibrated_method("farfan_core.utils.validation.golden_rule.GoldenRuleValidator.assert_atomic_context")
4037:     def assert_atomic_context(self, state_obj: object) -> None:
4038:         """Ensure copy-on-write semantics for per-step state."""
4039:
4040:         obj_id = id(state_obj)
4041:         if obj_id in self._state_ids:
4042:             raise GoldenRuleViolation("State object reused across steps")
4043:
4044:         self._state_ids.add(obj_id)
4045:
4046:     @calibrated_method("farfan_core.utils.validation.golden_rule.GoldenRuleValidator.assert_deterministic_dag")
4047:     def assert_deterministic_dag(self, step_ids: list[str]) -> None:
4048:         """Validate deterministic ordering and absence of cycles."""
4049:
4050:         if len(step_ids) != len(set(step_ids)):
4051:             raise GoldenRuleViolation("Duplicate step identifiers detected")
4052:
4053:         # Validate that step_ids is a subsequence of the canonical step catalog and in the same order
4054:         canonical = self._baseline_step_catalog
4055:         canonical_indices = {step_id: idx for idx, step_id in enumerate(canonical)}
4056:         try:
4057:             indices = [canonical_indices[step_id] for step_id in step_ids]
4058:         except KeyError as e:
4059:             raise GoldenRuleViolation(f"Step ID '{e.args[0]}' not found in canonical step catalog")
4060:         if indices != sorted(indices):
4061:             raise GoldenRuleViolation("Execution chain deviates from canonical order")
4062:     @calibrated_method("farfan_core.utils.validation.golden_rule.GoldenRuleValidator.assert_homogeneous_treatment")
4063:     def assert_homogeneous_treatment(self, predicate_set: Iterable[str]) -> None:
4064:         """Ensure identical predicate set is applied across all questions."""
4065:
4066:         fingerprint = {str(item) for item in predicate_set}
4067:
4068:         if self._predicate_signature is None:
4069:             self._predicate_signature = fingerprint
4070:             return
4071:
4072:         if fingerprint != self._predicate_signature:
4073:             raise GoldenRuleViolation("Predicate set mismatch detected")
4074:
4075:     @property
4076:     @calibrated_method("farfan_core.utils.validation.golden_rule.GoldenRuleValidator.baseline_step_catalog")
4077:     def baseline_step_catalog(self) -> list[str]:
4078:         """Expose the baseline step catalog for downstream validation."""
4079:
4080:         return list(self._baseline_step_catalog)
4081:
```

```
4082:
4083:
4084: ================================================================================
4085: FILE: src/farfan_pipeline/utils/validation/predicates.py
4086: ================================================================================
4087:
4088: #!/usr/bin/env python3
4089: """
4090: Validation Predicates - Precondition Checks for Execution
4091: ============================================================
4092:
4093: Provides reusable predicates for validating preconditions before
4094: executing analysis steps.
4095:
4096: Author: Integration Team - Agent 3
4097: Version: 1.0.0
4098: Python: 3.10+
4099: """
4100:
4101: from dataclasses import dataclass
4102: from typing import Any
4103:
4104:
4105: @dataclass
4106: class ValidationResult:
4107:     """Result of a validation check."""
4108:
4109:     is_valid: bool
4110:     severity: str  # ERROR, WARNING, INFO
4111:     message: str
4112:     context: dict[str, Any]
4113:
4114:
4115: class ValidationPredicates:
4116:     """
4117:     Collection of validation predicates for precondition checking.
4118:
4119:     These predicates verify that all required preconditions are met
4120:     before executing specific analysis steps.
4121:     """
4122:
4123:     @staticmethod
4124:     def verify_scoring_preconditions(
4125:         question_spec: dict[str, Any], execution_results: dict[str, Any], plan_text: str
4126:     ) -> ValidationResult:
4127:         """
4128:         Verify preconditions for TYPE_A scoring modality.
4129:
4130:         PRECONDITIONS for TYPE_A (Binary presence/absence):
4131:         - question_spec must have expected_elements list
4132:         - execution_results must be non-empty dict
4133:         - plan_text must be non-empty string
4134:
4135:         Args:
4136:             question_spec: Question specification from rubric
4137:             execution_results: Results from execution pipeline
```

```
4138:             plan_text: Full plan document text
4139:
4140:         Returns:
4141:             ValidationResult indicating if preconditions are met
4142:         """
4143:         errors = []
4144:
4145:         # Check question_spec has expected_elements
4146:         if not isinstance(question_spec, dict):
4147:             errors.append("question_spec must be a dictionary")
4148:         elif "expected_elements" not in question_spec:
4149:             errors.append("question_spec must have 'expected_elements' field")
4150:         elif not isinstance(question_spec.get("expected_elements"), list):
4151:             errors.append("expected_elements must be a list")
4152:         elif len(question_spec.get("expected_elements", [])) == 0:
4153:             errors.append("expected_elements cannot be empty")
4154:
4155:         # Check execution_results
4156:         if not isinstance(execution_results, dict):
4157:             errors.append("execution_results must be a dictionary")
4158:         elif len(execution_results) == 0:
4159:             errors.append("execution_results cannot be empty")
4160:
4161:         # Check plan_text
4162:         if not isinstance(plan_text, str):
4163:             errors.append("plan_text must be a string")
4164:         elif len(plan_text.strip()) == 0:
4165:             errors.append("plan_text cannot be empty")
4166:
4167:         if errors:
4168:             return ValidationResult(
4169:                 is_valid=False,
4170:                 severity="ERROR",
4171:                 message="; ".join(errors),
4172:                 context={
4173:                     "question_id": question_spec.get("id", "UNKNOWN"),
4174:                     "errors": errors,
4175:                 },
4176:             )
4177:
4178:         return ValidationResult(
4179:             is_valid=True,
4180:             severity="INFO",
4181:             message="All scoring preconditions met",
4182:             context={
4183:                 "question_id": question_spec.get("id"),
4184:                 "expected_elements_count": len(
4185:                     question_spec.get("expected_elements", [])
4186:                 ),
4187:                 "execution_results_keys": list(execution_results.keys()),
4188:             },
4189:         )
4190:
4191:     @staticmethod
4192:     def _validate_expected_elements_types(
4193:         question_schema: list[Any] | dict[str, Any] | None,
```

```
4194:         chunk_schema: list[Any] | dict[str, Any] | None,
4195:         question_id: str,
4196:     ) -> ValidationResult:
4197:         """
4198:         Validate that question_schema and chunk_schema have valid types.
4199:
4200:         Args:
4201:             question_schema: Schema object (None, list, or dict)
4202:             chunk_schema: Schema object (None, list, or dict)
4203:             question_id: Question identifier for error context
4204:
4205:         Returns:
4206:             ValidationResult indicating if schema types are valid
4207:
4208:         Raises:
4209:             TypeError: If either schema has invalid type (not None, list, or dict)
4210:             ValueError: If schemas are both lists with different lengths or both dicts with different keys
4211:         """
4212:         if question_schema is None and chunk_schema is None:
4213:             return ValidationResult(
4214:                 is_valid=True,
4215:                 severity="INFO",
4216:                 message=f"Question {question_id} has valid schema types",
4217:                 context={
4218:                     "question_id": question_id,
4219:                     "question_schema_type": "None",
4220:                     "chunk_schema_type": "None",
4221:                 },
4222:             )
4223:
4224:         errors = []
4225:
4226:         if question_schema is not None and not isinstance(question_schema, list | dict):
4227:             errors.append("question_schema has invalid type (not None, list, or dict)")
4228:
4229:         if chunk_schema is not None and not isinstance(chunk_schema, list | dict):
4230:             errors.append("chunk_schema has invalid type (not None, list, or dict)")
4231:
4232:         if errors:
4233:             raise TypeError(
4234:                 f"Question {question_id} has invalid schema types: {'; '.join(errors)}"
4235:             )
4236:
4237:         if (
4238:             isinstance(question_schema, list)
4239:             and isinstance(chunk_schema, list)
4240:             and len(question_schema) != len(chunk_schema)
4241:         ):
4242:             raise ValueError(
4243:                 f"Question {question_id} schema length mismatch: "
4244:                 f"question_schema has {len(question_schema)} elements, "
4245:                 f"chunk_schema has {len(chunk_schema)} elements"
4246:             )
4247:
4248:         if isinstance(question_schema, dict) and isinstance(chunk_schema, dict):
4249:             question_keys = set(question_schema.keys())
```

```
4250:             chunk_keys = set(chunk_schema.keys())
4251:             key_diff = question_keys.symmetric_difference(chunk_keys)
4252:             if key_diff:
4253:                 sorted_diff = sorted(key_diff)
4254:                 raise ValueError(
4255:                     f"Question {question_id} schema key mismatch: "
4256:                     f"symmetric difference in keys: {sorted_diff}"
4257:                 )
4258:
4259:         question_type = (
4260:             "None"
4261:             if question_schema is None
4262:             else "list" if isinstance(question_schema, list) else "dict"
4263:         )
4264:         chunk_type = (
4265:             "None"
4266:             if chunk_schema is None
4267:             else "list" if isinstance(chunk_schema, list) else "dict"
4268:         )
4269:
4270:         return ValidationResult(
4271:             is_valid=True,
4272:             severity="INFO",
4273:             message=f"Question {question_id} has valid schema types",
4274:             context={
4275:                 "question_id": question_id,
4276:                 "question_schema_type": question_type,
4277:                 "chunk_schema_type": chunk_type,
4278:             },
4279:         )
4280:
4281:     @staticmethod
4282:     def _validate_element_compatibility(
4283:         question_elements: list[dict[str, Any]],
4284:         chunk_elements: list[dict[str, Any]],
4285:         question_id: str,
4286:     ) -> int:
4287:         """
4288:         Validate compatibility between question elements and chunk elements.
4289:
4290:         Validates type, required, and minimum fields for all element pairs.
4291:
4292:         Args:
4293:             question_elements: List of element dictionaries from question schema
4294:             chunk_elements: List of element dictionaries from chunk schema
4295:             question_id: Question identifier for error context
4296:
4297:         Returns:
4298:             Count of successfully validated element pairs
4299:
4300:         Raises:
4301:             ValueError: If element validation fails (type mismatch, required field missing,
4302:                         or minimum threshold not met)
4303:         """
4304:         if not question_elements and not chunk_elements:
4305:             return 0
```

```
4306:
4307:         if len(question_elements) != len(chunk_elements):
4308:             raise ValueError(
4309:                 f"Question {question_id}: element count mismatch - "
4310:                 f"question has {len(question_elements)} elements, "
4311:                 f"chunk has {len(chunk_elements)} elements"
4312:             )
4313:
4314:         validated_count = 0
4315:
4316:         for idx, (q_elem, c_elem) in enumerate(
4317:             zip(question_elements, chunk_elements, strict=True)
4318:         ):
4319:             if not isinstance(q_elem, dict) or not isinstance(c_elem, dict):
4320:                 raise ValueError(
4321:                     f"Question {question_id}: element at index {idx} must be a dictionary"
4322:                 )
4323:
4324:             # Type validation
4325:             q_type = q_elem.get("type")
4326:             c_type = c_elem.get("type")
4327:             if q_type and c_type and q_type != c_type:
4328:                 raise ValueError(
4329:                     f"Question {question_id}: element type mismatch at index {idx} - "
4330:                     f"question expects type '{q_type}', chunk has type '{c_type}'"
4331:                 )
4332:
4333:             # Required field validation
4334:             q_required = q_elem.get("required", False)
4335:             c_required = c_elem.get("required", False)
4336:             if q_required and not c_required:
4337:                 raise ValueError(
4338:                     f"Question {question_id}: required field mismatch at index {idx} - "
4339:                     f"question requires element but chunk does not"
4340:                 )
4341:
4342:             # Minimum field validation
4343:             q_minimum = q_elem.get("minimum", 0)
4344:             c_minimum = c_elem.get("minimum", 0)
4345:             if c_minimum < q_minimum:
4346:                 element_type = q_elem.get("type", "unknown")
4347:                 raise ValueError(
4348:                     f"Question {question_id}: minimum threshold not met at index {idx} - "
4349:                     f"element type '{element_type}' requires minimum {q_minimum}, "
4350:                     f"but chunk has minimum {c_minimum}"
4351:                 )
4352:
4353:             validated_count += 1
4354:
4355:         return validated_count
4356:
4357:     @staticmethod
4358:     def verify_expected_elements(
4359:         question_spec: dict[str, Any], cuestionario_data: dict[str, Any] | None = None
4360:     ) -> ValidationResult:
4361:         """
```

```
        Verify that expected_elements are defined correctly.

        Args:
            question_spec: Question specification from rubric
            cuestionario_data: Full cuestionario metadata

        Returns:
            ValidationResult indicating if expected_elements are valid
        """
        question_id = question_spec.get("id", "UNKNOWN")

        expected_elements = question_spec.get("expected_elements")
        question_schema_raw = expected_elements
        chunk_schema_raw = question_spec.get("chunk_schema")

        if question_schema_raw is None:
            question_type = "None"
        elif isinstance(question_schema_raw, list):
            question_type = "list"
        elif isinstance(question_schema_raw, dict):
            question_type = "dict"
        else:
            question_type = "invalid"

        if chunk_schema_raw is None:
            chunk_type = "None"
        elif isinstance(chunk_schema_raw, list):
            chunk_type = "list"
        elif isinstance(chunk_schema_raw, dict):
            chunk_type = "dict"
        else:
            chunk_type = "invalid"

        schema_types = (question_type, chunk_type)

        ValidationPredicates._validate_expected_elements_types(
            question_schema_raw, chunk_schema_raw, question_id
        )

        if question_type == "None":
            return ValidationResult(
                is_valid=False,
                severity="WARNING",
                message=f"Question {question_id} has no expected_elements defined",
                context={"question_id": question_id, "schema_types": schema_types},
            )

        if (
            question_type == "list"
            and expected_elements is not None
            and len(expected_elements) == 0
        ):
            return ValidationResult(
                is_valid=False,
                severity="WARNING",
                message=f"Question {question_id} has empty expected_elements",
```

```
                context={"question_id": question_id, "schema_types": schema_types},
            )

        # Validate element compatibility if both schemas are lists of dicts
        validated_element_count = 0
        if (
            question_type == "list"
            and chunk_type == "list"
            and expected_elements
            and all(isinstance(e, dict) for e in expected_elements)
        ):
            chunk_elements = (
                chunk_schema_raw if isinstance(chunk_schema_raw, list) else []
            )
            if chunk_elements and all(isinstance(e, dict) for e in chunk_elements):
                try:
                    validated_element_count = (
                        ValidationPredicates._validate_element_compatibility(
                            expected_elements, chunk_elements, question_id
                        )
                    )
                except ValueError as e:
                    return ValidationResult(
                        is_valid=False,
                        severity="ERROR",
                        message=str(e),
                        context={
                            "question_id": question_id,
                            "schema_types": schema_types,
                        },
                    )

        element_count = None
        if question_type == "list" and expected_elements is not None:
            element_count = len(expected_elements)

        return ValidationResult(
            is_valid=True,
            severity="INFO",
            message=f"Question {question_id} has valid expected_elements",
            context={
                "question_id": question_id,
                "expected_elements": expected_elements,
                "count": element_count,
                "schema_types": schema_types,
                "validated_element_count": validated_element_count,
            },
        )

    @staticmethod
    def verify_execution_context(
        question_id: str, policy_area: str, dimension: str
    ) -> ValidationResult:
        """
        Verify execution context parameters are valid.

```

```
4474:         Args:
4475:             question_id: Canonical question ID (P#-D#-Q#)
4476:             policy_area: Policy area (P1-P10)
4477:             dimension: Dimension (D1-D6)
4478:
4479:         Returns:
4480:             ValidationResult indicating if context is valid
4481:         """
4482:         errors = []
4483:
4484:         # Validate question_id format
4485:         if not question_id or not isinstance(question_id, str):
4486:             errors.append("question_id must be a non-empty string")
4487:         elif not question_id.startswith("P"):
4488:             errors.append(f"question_id '{question_id}' must start with 'P'")
4489:
4490:         # Validate policy_area
4491:         if not policy_area or not isinstance(policy_area, str):
4492:             errors.append("policy_area must be a non-empty string")
4493:         elif not policy_area.startswith("P"):
4494:             errors.append(f"policy_area '{policy_area}' must start with 'P'")
4495:         else:
4496:             try:
4497:                 area_num = int(policy_area[1:])
4498:                 if not (1 <= area_num <= 10):
4499:                     errors.append(f"policy_area '{policy_area}' must be P1-P10")
4500:             except ValueError:
4501:                 errors.append(f"Invalid policy_area format: '{policy_area}'")
4502:
4503:         # Validate dimension
4504:         if not dimension or not isinstance(dimension, str):
4505:             errors.append("dimension must be a non-empty string")
4506:         elif not dimension.startswith("D"):
4507:             errors.append(f"dimension '{dimension}' must start with 'D'")
4508:         else:
4509:             try:
4510:                 dim_num = int(dimension[1:])
4511:                 if not (1 <= dim_num <= 6):
4512:                     errors.append(f"dimension '{dimension}' must be D1-D6")
4513:             except ValueError:
4514:                 errors.append(f"Invalid dimension format: '{dimension}'")
4515:
4516:         if errors:
4517:             return ValidationResult(
4518:                 is_valid=False,
4519:                 severity="ERROR",
4520:                 message="; ".join(errors),
4521:                 context={
4522:                     "question_id": question_id,
4523:                     "policy_area": policy_area,
4524:                     "dimension": dimension,
4525:                     "errors": errors,
4526:                 },
4527:             )
4528:
4529:         return ValidationResult(
```

```
4530:             is_valid=True,
4531:             severity="INFO",
4532:             message="Execution context is valid",
4533:             context={
4534:                 "question_id": question_id,
4535:                 "policy_area": policy_area,
4536:                 "dimension": dimension,
4537:             },
4538:         )
4539:
4540:     @staticmethod
4541:     def verify_producer_availability(
4542:         producer_name: str, producers_dict: dict[str, Any]
4543:     ) -> ValidationResult:
4544:         """
4545:         Verify that a producer module is available and initialized.
4546:
4547:         Args:
4548:             producer_name: Name of the producer (e.g., 'derek_beach')
4549:             producers_dict: Dictionary of initialized producers
4550:
4551:         Returns:
4552:             ValidationResult indicating if producer is available
4553:         """
4554:         if producer_name not in producers_dict:
4555:             return ValidationResult(
4556:                 is_valid=False,
4557:                 severity="ERROR",
4558:                 message=f"Producer '{producer_name}' not found in initialized producers",
4559:                 context={
4560:                     "producer_name": producer_name,
4561:                     "available_producers": list(producers_dict.keys()),
4562:                 },
4563:             )
4564:
4565:         producer = producers_dict[producer_name]
4566:
4567:         # Check if producer is initialized
4568:         if isinstance(producer, dict):
4569:             status = producer.get("status")
4570:             if status != "initialized":
4571:                 return ValidationResult(
4572:                     is_valid=False,
4573:                     severity="ERROR",
4574:                     message=f"Producer '{producer_name}' status is '{status}'",
4575:                     context={
4576:                         "producer_name": producer_name,
4577:                         "status": status,
4578:                         "error": producer.get("error"),
4579:                     },
4580:                 )
4581:
4582:         return ValidationResult(
4583:             is_valid=True,
4584:             severity="INFO",
4585:             message=f"Producer '{producer_name}' is available and initialized",
```

```
4586:             context={"producer_name": producer_name, "status": "initialized"},
4587:         )
4588:
4589:
```