

```

        years.extend(int(y) for y in year_matches)

# Also check strategic_context for temporal_horizon
if hasattr(smart_chunk, 'strategic_context') and smart_chunk.strategic_context:
    ctx = smart_chunk.strategic_context
    if hasattr(ctx, 'temporal_horizon'):
        periods.append(ctx.temporal_horizon)

return TimeFacet(
    years=sorted(set(years))[:10], # Unique and sorted
    periods=periods[:5]
)

@calibrated_method("saaaaaa.processing.spc_ingestion.converter.SmartChunkConverter._extract_geo_facets")
def _extract_geo_facets(self, smart_chunk: Any) -> GeoFacet:
    """Extract geographic information from SPC strategic_context."""
    territories = []
    regions = []

    if hasattr(smart_chunk, 'strategic_context') and smart_chunk.strategic_context:
        ctx = smart_chunk.strategic_context
        if hasattr(ctx, 'geographic_scope'):
            territories.append(ctx.geographic_scope)
        # Could also parse from policy_entities with location types

    return GeoFacet(
        territories=territories[:5],
        regions=regions[:5]
    )

@calibrated_method("saaaaaa.processing.spc_ingestion.converter.SmartChunkConverter._build_provenance")
def _build_provenance(self, smart_chunk: Any) -> ProvenanceMap:
    """Build provenance from SPC metadata."""
    # Extract section info from section_hierarchy
    source_section = None
    if hasattr(smart_chunk, 'section_hierarchy') and smart_chunk.section_hierarchy:
        source_section = " > ".join(smart_chunk.section_hierarchy[:3])

    return ProvenanceMap(
        source_page=None, # SPC doesn't track page numbers
        source_section=source_section,
        extraction_method="smart_policy_chunking_v3.0"
    )

@calibrated_method("saaaaaa.processing.spc_ingestion.converter.SmartChunkConverter._extract_entities")
def _extract_entities(self, smart_chunk: Any) -> list[Entity]:
    """Extract entities from SPC policy_entities."""
    entities = []

    if hasattr(smart_chunk, 'policy_entities') and smart_chunk.policy_entities:
        for pe in smart_chunk.policy_entities[:20]: # Limit to 20
            entity = Entity(
                text=pe.text if hasattr(pe, 'text') else str(pe),
                entity_type=pe.entity_type if hasattr(pe, 'entity_type') else
'unknown',
                confidence=pe.confidence if hasattr(pe, 'confidence') else get_parameter_loader().get("saaaaaa.processing.spc_ingestion.converter.SmartChunkConverter._extract_entities").get("auto_param_L373_79", 0.8)
            )
            entities.append(entity)

    return entities

@calibrated_method("saaaaaa.processing.spc_ingestion.converter.SmartChunkConverter._extract_budget")

```

```

def _extract_budget(self, smart_chunk: Any) -> Budget | None:
    """
    Extract budget with comprehensive error handling and logging (H1.4).

    Implements 4 regex patterns and robust year extraction with 4 fallback strategies.
    """
    if not (hasattr(smart_chunk, 'strategic_context') and
    smart_chunk.strategic_context):
        return None

    ctx = smart_chunk.strategic_context
    if not (hasattr(ctx, 'budget_linkage') and ctx.budget_linkage):
        return None

    import re
    budget_text = ctx.budget_linkage

    # H1.4: 4 regex patterns for robust budget extraction
    patterns = [
        # Pattern 1: Currency symbol with optional scale
        r'[\$]?s*([0-9,]+(?:\.[0-9]+)?)s*(millones|mil millones|billion|billones)?',
        # Pattern 2: "X millones de pesos"
        r'(\d+(?:\.[0-9]+)?)s*millones?s+de\s+pesos',
        # Pattern 3: COP currency code
        r'COPs*\$?s*([0-9,]+(?:\.[0-9]+)?)',
        # Pattern 4: "presupuesto de $X"
        r'presupuesto\s+de\s+\$?s*([0-9,]+(?:\.[0-9]+)?)',
    ]

    amount = None
    scale_multiplier = 1

    for pattern in patterns:
        match = re.search(pattern, budget_text, re.IGNORECASE)
        if match:
            try:
                amount_str = match.group(1).replace(',', '')
                amount = float(amount_str)

                # Detect scale from second group or text
                scale_text = budget_text.lower()
                if 'millones' in scale_text or 'million' in scale_text:
                    scale_multiplier = 1_000_000
                elif 'mil millones' in scale_text or 'billion' in scale_text or
                'billones' in scale_text:
                    scale_multiplier = 1_000_000_000

                amount *= scale_multiplier
                break # Stop at first match

            except (ValueError, IndexError) as e:
                self.logger.warning(f"Budget pattern matched but parsing failed: {e}")
                continue

    if amount is None:
        return None

    # H1.4: Extract year with 4 fallback strategies
    year = self._extract_budget_year(budget_text, smart_chunk)

    # Build Budget object
    use = ctx.policy_intent if hasattr(ctx, 'policy_intent') else "General"

    self.logger.debug(
        f"Extracted budget: amount={amount:,.2f}, year={year}, "
        f"source='Strategic Context', use='{use[:30]}'"
    )

```

```

    return Budget(
        source="Strategic Context",
        use=use,
        amount=amount,
        year=year,
        currency="COP"
    )

@calibrated_method("saaaaaa.processing.spc_ingestion.converter.SmartChunkConverter._extract_budget_year")
def _extract_budget_year(self, budget_text: str, smart_chunk: Any) -> int:
    """
    Extract budget year with 4 fallback strategies (H1.4).

    Strategy 1: Extract from budget_text itself
    Strategy 2: Extract from temporal_dynamics markers
    Strategy 3: Use temporal_horizon from strategic_context
    Strategy 4: Default to 2024
    """
    import re

    # Strategy 1: Look for year in budget_text
    year_match = re.search(r'\b(202[0-9]|203[0-9])\b', budget_text)
    if year_match:
        return int(year_match.group(1))

    # Strategy 2: Check temporal_dynamics markers
    if hasattr(smart_chunk, 'temporal_dynamics') and smart_chunk.temporal_dynamics:
        temp = smart_chunk.temporal_dynamics
        if hasattr(temp, 'temporal_markers') and temp.temporal_markers:
            for marker in temp.temporal_markers:
                marker_text = marker[0] if isinstance(marker, (list, tuple)) else str(marker)
                year_match = re.search(r'\b(202[0-9]|203[0-9])\b', marker_text)
                if year_match:
                    return int(year_match.group(1))

    # Strategy 3: Check strategic_context temporal_horizon
    if hasattr(smart_chunk, 'strategic_context') and smart_chunk.strategic_context:
        ctx = smart_chunk.strategic_context
        if hasattr(ctx, 'temporal_horizon') and ctx.temporal_horizon:
            horizon_match = re.search(r'\b(202[0-9]|203[0-9])\b',
                ctx.temporal_horizon)
            if horizon_match:
                return int(horizon_match.group(1))

    # Strategy 4: Default to 2024
    self.logger.debug("No year found in budget context, defaulting to 2024")
    return 2024

@calibrated_method("saaaaaa.processing.spc_ingestion.converter.SmartChunkConverter._extract_kpi")
def _extract_kpi(self, smart_chunk: Any) -> KPI | None:
    """Extract KPI if chunk contains indicator information."""
    # Check if chunk_type suggests this is a metric
    chunk_type_str = smart_chunk.chunk_type.value if hasattr(smart_chunk.chunk_type, 'value') else str(smart_chunk.chunk_type)

    if 'METRICA' in chunk_type_str.upper() or 'EVALUACION' in chunk_type_str.upper():
        # Try to extract indicator from text
        import re
        text = smart_chunk.text
        # Look for percentage targets (e.g., "90%", "meta del 85%")
        target_match = re.search(r'meta.*?([0-9]+(?:\.[0-9]+)?)\s*%', text, re.IGNORECASE)
        if target_match:
            return KPI(
                indicator_name=smart_chunk.text[:80], # Use chunk text as indicator

```

```

name
    target_value=float(target_match.group(1)),
    unit="%",
    year=None
)

return None

def _calculate_quality_metrics(
    self,
    smart_chunks: list[Any],
    chunk_graph: ChunkGraph
) -> QualityMetrics:
    """Calculate quality metrics from SPC data."""
    # Provenance completeness
    chunks_with_provenance = sum(
        1 for c in chunk_graph.chunks.values()
        if c.provenance and c.provenance.source_section
    )
    provenance_completeness = chunks_with_provenance / len(chunk_graph.chunks) if
chunk_graph.chunks else get_parameter_loader().get("saaaaaa.processing.spc_ingestion.conve
rter.SmartChunkConverter._extract_kpi").get("auto_param_L526_110", 0.0)

    # Average coherence from SPC coherence_score
    avg_coherence = sum(sc.coherence_score for sc in smart_chunks) / len(smart_chunks)
if smart_chunks else get_parameter_loader().get("saaaaaa.processing.spc_ingestion.conve
rter.SmartChunkConverter._extract_kpi").get("auto_param_L529_112", 0.0)

    # Average completeness from SPC completeness_index
    avg_completeness = sum(sc.completeness_index for sc in smart_chunks) /
len(smart_chunks) if smart_chunks else get_parameter_loader().get("saaaaaa.processing.spc
ingestion.converter.SmartChunkConverter._extract_kpi").get("auto_param_L532_118", 0.0)

    # Budget consistency
    chunks_with_budget = sum(1 for c in chunk_graph.chunks.values() if c.budget)
    budget_consistency = chunks_with_budget / len(chunk_graph.chunks) if
chunk_graph.chunks else get_parameter_loader().get("saaaaaa.processing.spc_ingestion.conve
rter.SmartChunkConverter._extract_kpi").get("auto_param_L536_101", 0.0)

    # Temporal robustness
    chunks_with_time = sum(1 for c in chunk_graph.chunks.values() if
c.time_facets.years)
    temporal_robustness = chunks_with_time / len(chunk_graph.chunks) if
chunk_graph.chunks else get_parameter_loader().get("saaaaaa.processing.spc_ingestion.conve
rter.SmartChunkConverter._extract_kpi").get("auto_param_L540_100", 0.0)

    # Chunk context coverage (from edges)
    chunks_with_edges = len({e[0] for e in chunk_graph.edges} | {e[1] for e in
chunk_graph.edges})
    chunk_context_coverage = chunks_with_edges / len(chunk_graph.chunks) if
chunk_graph.chunks else get_parameter_loader().get("saaaaaa.processing.spc_ingestion.conve
rter.SmartChunkConverter._extract_kpi").get("auto_param_L544_104", 0.0)

    return QualityMetrics(
        provenance_completeness=provenance_completeness,
        structural_consistency=avg_coherence,
        boundary_f1=avg_completeness,
        kpi_linkage_rate=get_parameter_loader().get("saaaaaa.processing.spc_ingestion.
converter.SmartChunkConverter._extract_kpi").get("auto_param_L550_29", 0.0), # Would need
KPI analysis
        budget_consistency_score=budget_consistency,
        temporal_robustness=temporal_robustness,
        chunk_context_coverage=chunk_context_coverage
    )

def _generate_integrity_index(
    self,
    chunk_hashes: dict[str, str],

```

```

    document_metadata: dict[str, Any]
) -> IntegrityIndex:
    """
    Generate cryptographic integrity index.

    Uses BLAKE2b-256 to compute aggregate hash of all chunk hashes.
    NOT a true Merkle tree - simply hashes sorted JSON representation.
    """
    # Generate root hash from all chunk hashes (sorted for determinism)
    combined = json.dumps(chunk_hashes, sort_keys=True).encode('utf-8')
    blake2b_root = hashlib.blake2b(combined, digest_size=32).hexdigest()

    return IntegrityIndex(
        blake2b_root=blake2b_root,
        chunk_hashes=chunk_hashes
    )

def _preserve_spc_rich_data(
    self,
    smart_chunks: list[Any],
    document_metadata: dict[str, Any]
) -> dict[str, Any]:
    """
    Preserve SPC rich data in metadata for executor access.

    This is critical for enabling executors to access the full SPC analysis:
    - Embeddings (semantic, policy, causal, temporal)
    - Causal chains with evidence
    - Strategic context
    - Quality scores
    """
    enriched = dict(document_metadata)

    # Store serializable SPC data
    spc_rich_data = {}

    for sc in smart_chunks:
        chunk_data = {
            'chunk_id': sc.chunk_id,
            'semantic_density': sc.semantic_density,
            'coherence_score': sc.coherence_score,
            'completeness_index': sc.completeness_index,
            'strategic_importance': sc.strategic_importance,
            'information_density': sc.information_density,
            'actionability_score': sc.actionability_score,
        }

        # Add embeddings if available (as lists for JSON serialization)
        # CRITICAL: Fail-fast if embeddings cannot be preserved (no silent data loss)
        if hasattr(sc, 'semantic_embedding') and sc.semantic_embedding is not None:
            try:
                import numpy as np
            except ImportError as e:
                self.logger.error(
                    f"Chunk {sc.chunk_id}: NumPy is required for embedding"
                    preservation but not available"
                )
                raise RuntimeError(
                    "NumPy is required for SPC embedding preservation. "
                    "Install with: pip install numpy>=1.26.0"
                ) from e

        # Validate embedding type
        if not isinstance(sc.semantic_embedding, np.ndarray):
            self.logger.error(
                f"Chunk {sc.chunk_id}: semantic_embedding is not np.ndarray, "
                f"got {type(sc.semantic_embedding)}"
            )

```

```

        raise TypeError(
            f"Expected semantic_embedding to be np.ndarray, got {type(sc.semantic_embedding)}")
    )

    # Convert to list for JSON serialization
    try:
        chunk_data['semantic_embedding'] = sc.semantic_embedding.tolist()
        chunk_data['embedding_dim'] = sc.semantic_embedding.shape[0]
        self.logger.debug(
            f"Chunk {sc.chunk_id}: Preserved embedding with dimension {sc.semantic_embedding.shape[0]}"
        )
    except (AttributeError, IndexError) as e:
        self.logger.error(
            f"Chunk {sc.chunk_id}: Failed to convert embedding to list: {e}"
        )
        raise RuntimeError(
            f"Embedding conversion failed for chunk {sc.chunk_id}: {e}"
        ) from e

    # Add causal chain summary
    if hasattr(sc, 'causal_chain') and sc.causal_chain:
        chunk_data['causal_chain_count'] = len(sc.causal_chain)
        chunk_data['causal_evidence'] = [
            {
                'dimension': ce.dimension if hasattr(ce, 'dimension') else
'unknown',
                'confidence': ce.confidence if hasattr(ce, 'confidence') else get_
parameter_loader().get("saaaaaa.processing.spc_ingestion.converter.SmartChunkConverter._ex
tract_kpi").get("auto_param_L651_86", 0.0),
            }
            for ce in sc.causal_chain[:5] # Top 5
        ]

    # Add strategic context summary
    if hasattr(sc, 'strategic_context') and sc.strategic_context:
        ctx = sc.strategic_context
        chunk_data['strategic_context'] = {
            'policy_intent': ctx.policy_intent if hasattr(ctx, 'policy_intent')
else None,
            'implementation_phase': ctx.implementation_phase if hasattr(ctx,
'implementation_phase') else None,
            'temporal_horizon': ctx.temporal_horizon if hasattr(ctx,
'temporal_horizon') else None,
        }

    # Add topic distribution
    if hasattr(sc, 'topic_distribution') and sc.topic_distribution:
        chunk_data['topic_distribution'] = dict(sc.topic_distribution)

    spc_rich_data[sc.chunk_id] = chunk_data

    enriched['spc_rich_data'] = spc_rich_data
    enriched['spc_version'] = 'SMART-CHUNK-3.0-FINAL'
    enriched['conversion_timestamp'] = document_metadata.get('processing_timestamp',
'unknown')

    self.logger.info(f"Preserved rich SPC data for {len(spc_rich_data)} chunks")

    return enriched

```

===== FILE: src/saaaaaa/processing/spc\_ingestion/quality\_gates.py =====

"""

Quality gates for SPC (Smart Policy Chunks) validation - Canonical Phase-One.

Validates quality metrics, enforces invariants, and ensures compatibility  
with downstream phases in the canonical pipeline flux.

MAXIMUM STANDARD: No tolerance for data quality degradation.  
"""

```
import logging
from pathlib import Path
from typing import Any
from saaaaa import get_parameter_loader
from saaaaa.core.calibration.decorators import calibrated_method
```

```
logger = logging.getLogger(__name__)
```

```
class SPCQualityGates:
```

```
    """Quality validation gates for Smart Policy Chunks ingestion."""
```

```
    # Phase-one output quality thresholds
```

```
    MIN_CHUNKS = 5
```

```
    MAX_CHUNKS = 200
```

```
    MIN_CHUNK_LENGTH = 50 # characters
```

```
    MAX_CHUNK_LENGTH = 5000
```

```
    MIN_STRATEGIC_SCORE = 0.3
```

```
    MIN_QUALITY_SCORE = 0.5
```

```
    REQUIRED_CHUNK_FIELDS = ['text', 'chunk_id', 'strategic_importance', 'quality_score']
```

```
    # Compatibility thresholds for downstream phases
```

```
    MIN_EMBEDDING_DIM = 384 # For semantic analysis
```

```
    REQUIRED_METADATA_FIELDS = ['document_id', 'title', 'version']
```

```
    # CRITICAL quality metrics (per README specifications)
```

```
    MIN_PROVENANCE_COMPLETENESS = 1.0 # 100% REQUIRED (no partial coverage tolerated)
```

```
    MIN_STRUCTURAL_CONSISTENCY = 1.0 # 100% REQUIRED (perfect structure)
```

```
    MIN_BOUNDARY_F1 = 0.85 # Chunk boundary quality
```

```
    MIN_BUDGET_CONSISTENCY = 0.95 # Budget data consistency
```

```
    MIN_TEMPORAL_ROBUSTNESS = 0.80 # Temporal data quality
```

```
    @calibrated_method("saaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_input")
```

```
    def validate_input(self, document_path: Path) -> dict[str, Any]:
```

```
        """
```

```
        Validate input document before processing.
```

```
        Args:
```

```
            document_path: Path to input document
```

```
        Returns:
```

```
            Dictionary with validation results
```

```
        """
```

```
        failures = []
```

```
        # Check file exists
```

```
        if not document_path.exists():
```

```
            failures.append(f"Input document not found: {document_path}")
```

```
            return {"passed": False, "failures": failures}
```

```
        # Check file size (not empty, not too large)
```

```
        file_size = document_path.stat().st_size
```

```
        if file_size == 0:
```

```
            failures.append("Input document is empty")
```

```
        elif file_size > 50 * 1024 * 1024: # 50MB limit
```

```
            failures.append(f"Input document too large: {file_size / 1024 / 1024:.1f}MB")
```

```
        # Check file extension
```

```
        if document_path.suffix.lower() not in ['.txt', '.pdf', '.json']:
```

```
            failures.append(f"Unsupported file type: {document_path.suffix}")
```

```
        return {
```

```
            "passed": len(failures) == 0,
```

```

        "failures": failures,
        "file_size_bytes": file_size,
    }

@calibrated_method("saaaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_chunks")
def validate_chunks(self, chunks: list[dict[str, Any]]) -> dict[str, Any]:
    """
    Validate processed chunks from phase-one.

    Args:
        chunks: List of smart policy chunks

    Returns:
        Dictionary with validation results
    """
    failures = []
    warnings = []

    # Check chunk count
    if len(chunks) < self.MIN_CHUNKS:
        failures.append(f"Too few chunks: {len(chunks)} < {self.MIN_CHUNKS}")
    elif len(chunks) > self.MAX_CHUNKS:
        warnings.append(f"High chunk count: {len(chunks)} > {self.MAX_CHUNKS}")

    # Validate each chunk
    for idx, chunk in enumerate(chunks):
        chunk_id = chunk.get('chunk_id', f'chunk_{idx}')

        # Check required fields
        for field in self.REQUIRED_CHUNK_FIELDS:
            if field not in chunk:
                failures.append(f"{chunk_id}: Missing required field '{field}'")

        # Check chunk text length
        text = chunk.get('text', "")
        if len(text) < self.MIN_CHUNK_LENGTH:
            failures.append(f"{chunk_id}: Text too short: {len(text)} < {self.MIN_CHUNK_LENGTH}")
        elif len(text) > self.MAX_CHUNK_LENGTH:
            warnings.append(f"{chunk_id}: Text very long: {len(text)} > {self.MAX_CHUNK_LENGTH}")

        # Check quality scores
        strategic_score = chunk.get('strategic_importance', 0)
        if strategic_score < self.MIN_STRATEGIC_SCORE:
            warnings.append(f"{chunk_id}: Low strategic importance: {strategic_score}")

        quality_score = chunk.get('quality_score', 0)
        if quality_score < self.MIN_QUALITY_SCORE:
            warnings.append(f"{chunk_id}: Low quality score: {quality_score}")

    return {
        "passed": len(failures) == 0,
        "failures": failures,
        "warnings": warnings,
        "chunk_count": len(chunks),
    }

@calibrated_method("saaaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_output_compatibility")
def validate_output_compatibility(self, output: dict[str, Any]) -> dict[str, Any]:
    """
    Validate output structure for compatibility with downstream phases.

    Ensures the phase-one output can be consumed by next phases in the flux.
    """

```

Args:

output: Phase-one output dictionary

Returns:

Dictionary with validation results

```
failures = []
```

```
# Check required top-level keys
```

```
if 'chunks' not in output:
```

```
    failures.append("Missing 'chunks' in output")
```

```
if 'metadata' not in output:
```

```
    failures.append("Missing 'metadata' in output")
```

```
else:
```

```
    # Validate metadata fields
```

```
    metadata = output['metadata']
```

```
    for field in self.REQUIRED_METADATA_FIELDS:
```

```
        if field not in metadata:
```

```
            failures.append(f"Missing required metadata field: '{field}'")
```

```
# Check chunks structure
```

```
if 'chunks' in output:
```

```
    chunks_result = self.validate_chunks(output['chunks'])
```

```
    if not chunks_result['passed']:
```

```
        failures.extend(chunks_result['failures'])
```

```
return {
```

```
    "passed": len(failures) == 0,
```

```
    "failures": failures,
```

```
}
```

```
@calibrated_method("saaaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics")
```

```
def validate_quality_metrics(self, quality_metrics: Any) -> dict[str, Any]:
```

```
    Validate quality metrics from CanonPolicyPackage against MAXIMUM STANDARDS.
```

```
    Enforces strict thresholds per README specifications. No degradation tolerated.
```

Args:

quality\_metrics: QualityMetrics instance from CanonPolicyPackage

Returns:

Dictionary with validation results:

```
{
```

```
    "passed": bool,
```

```
    "failures": List[str], # CRITICAL failures that MUST be fixed
```

```
    "warnings": List[str], # Non-critical warnings
```

```
    "metrics": Dict[str, float] # Actual metric values
```

```
}
```

```
failures = []
```

```
warnings = []
```

```
metrics_dict = {}
```

```
# Extract metrics (handle both object and dict)
```

```
if hasattr(quality_metrics, '__dict__'):
```

```
    # It's an object
```

```
    provenance_completeness = getattr(quality_metrics, 'provenance_completeness',
```

```
get_parameter_loader().get("saaaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates
```

```
.validate_quality_metrics").get("auto_param_L193_90", 0.0))
```

```
    structural_consistency = getattr(quality_metrics, 'structural_consistency', ge
```

```
t_parameter_loader().get("saaaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates.v
```

```
alidate_quality_metrics").get("auto_param_L194_88", 0.0))
```

```
    boundary_f1 = getattr(quality_metrics, 'boundary_f1', get_parameter_loader().g
```

```
et("saaaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metric
```

```
s").get("auto_param_L195_66", 0.0))
```

```

        budget_consistency = getattr(quality_metrics, 'budget_consistency_score', get_
parameter_loader().get("saaaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics").get("auto_param_L196_86", 0.0))
        temporal_robustness = getattr(quality_metrics, 'temporal_robustness', get_parameter_loader().get("saaaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics").get("auto_param_L197_82", 0.0))
        chunk_context_coverage = getattr(quality_metrics, 'chunk_context_coverage', get_parameter_loader().get("saaaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics").get("auto_param_L198_88", 0.0))
    else:
        # It's a dict
        provenance_completeness = quality_metrics.get('provenance_completeness', get_parameter_loader().get("saaaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics").get("auto_param_L201_85", 0.0))
        structural_consistency = quality_metrics.get('structural_consistency', get_parameter_loader().get("saaaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics").get("auto_param_L202_83", 0.0))
        boundary_f1 = quality_metrics.get('boundary_f1', get_parameter_loader().get("saaaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics").get("auto_param_L203_61", 0.0))
        budget_consistency = quality_metrics.get('budget_consistency_score', get_parameter_loader().get("saaaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics").get("auto_param_L204_81", 0.0))
        temporal_robustness = quality_metrics.get('temporal_robustness', get_parameter_loader().get("saaaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics").get("auto_param_L205_77", 0.0))
        chunk_context_coverage = quality_metrics.get('chunk_context_coverage', get_parameter_loader().get("saaaaaa.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics").get("auto_param_L206_83", 0.0))

    # Store actual values
    metrics_dict = {
        'provenance_completeness': provenance_completeness,
        'structural_consistency': structural_consistency,
        'boundary_f1': boundary_f1,
        'budget_consistency_score': budget_consistency,
        'temporal_robustness': temporal_robustness,
        'chunk_context_coverage': chunk_context_coverage,
    }

    # CRITICAL: Provenance completeness MUST be 100%
    if provenance_completeness < self.MIN_PROVENANCE_COMPLETENESS:
        failures.append(
            f"● CRITICAL: Provenance completeness below threshold: "
            f"{provenance_completeness:.2%} < {self.MIN_PROVENANCE_COMPLETENESS:.0%}."
        )

        f"Every token must be traceable to source (README requirement)."
    )
    logger.error(
        f"Provenance completeness FAILED: {provenance_completeness:.2%} "
        f"(required: {self.MIN_PROVENANCE_COMPLETENESS:.0%})"
    )

    # CRITICAL: Structural consistency MUST be perfect
    if structural_consistency < self.MIN_STRUCTURAL_CONSISTENCY:
        failures.append(
            f"● CRITICAL: Structural consistency below threshold: "
            f"{structural_consistency:.2%} < {self.MIN_STRUCTURAL_CONSISTENCY:.0%}."
            f"Policy structure must be perfectly parsed (FASE 3 gate)."
        )
    logger.error(
        f"Structural consistency FAILED: {structural_consistency:.2%} "
        f"(required: {self.MIN_STRUCTURAL_CONSISTENCY:.0%})"
    )

    # HIGH: Boundary F1 for chunk quality
    if boundary_f1 < self.MIN_BOUNDARY_F1:
        failures.append(

```

```

        f"● HIGH: Boundary F1 below threshold: "
        f"{boundary_f1:.2f} < {self.MIN_BOUNDARY_F1}. "
        f"Chunk boundaries are not accurate enough (FASE 8 gate)."
    )
    logger.error(
        f"Boundary F1 FAILED: {boundary_f1:.2f} "
        f"(required: {self.MIN_BOUNDARY_F1})"
    )

# HIGH: Budget consistency for financial data
if budget_consistency < self.MIN_BUDGET_CONSISTENCY:
    warnings.append(
        f"● Budget consistency below threshold: "
        f"{budget_consistency:.2%} < {self.MIN_BUDGET_CONSISTENCY:.0%}. "
        f"Budget data may have inconsistencies (FASE 6 gate)."
    )
    logger.warning(
        f"Budget consistency WARNING: {budget_consistency:.2%} "
        f"(recommended: {self.MIN_BUDGET_CONSISTENCY:.0%})"
    )

# MEDIUM: Temporal robustness
if temporal_robustness < self.MIN_TEMPORAL_ROBUSTNESS:
    warnings.append(
        f"● Temporal robustness below threshold: "
        f"{temporal_robustness:.2%} < {self.MIN_TEMPORAL_ROBUSTNESS:.0%}. "
        f"Temporal data may be incomplete."
    )

# INFO: Chunk context coverage
if chunk_context_coverage < get_parameter_loader().get("saaaaaa.processing.spc_ing
estion.quality_gates.SPCQualityGates.validate_quality_metrics").get("auto_param_L275_36",
0.5):
    warnings.append(
        f"i Low chunk context coverage: {chunk_context_coverage:.2%}. "
        f"Few inter-chunk relationships detected."
    )

# Summary logging
if failures:
    logger.error(f"Quality metrics validation FAILED with {len(failures)} critical
issues")
elif warnings:
    logger.warning(f"Quality metrics validation PASSED with {len(warnings)}
warnings")
else:
    logger.info("Quality metrics validation PASSED - All thresholds met")

return {
    "passed": len(failures) == 0,
    "failures": failures,
    "warnings": warnings,
    "metrics": metrics_dict,
}

# Legacy alias for backwards compatibility
class QualityGates(SPCQualityGates):
    """Legacy alias for SPCQualityGates."""
    pass

===== FILE: src/saaaaaa/processing/spc_ingestion/structural.py =====
"""
Structural normalization with policy-awareness.

Segments documents into policy-aware units.
"""

```

```

from typing import Any
from saaaaaa.core.calibration.decorators import calibrated_method

class StructuralNormalizer:
    """Policy-aware structural normalizer."""

    @calibrated_method("saaaaaa.processing.spc_ingestion.structural.StructuralNormalizer.normalize")
    def normalize(self, raw_objects: dict[str, Any]) -> dict[str, Any]:
        """
        Normalize document structure with policy awareness.

        Args:
            raw_objects: Raw parsed objects

        Returns:
            Policy graph with structured sections
        """
        policy_graph = {
            "sections": [],
            "policy_units": [],
            "axes": [],
            "programs": [],
            "projects": [],
            "years": [],
            "territories": [],
        }

        # Extract sections from pages
        for page in raw_objects.get("pages", []):
            text = page.get("text", "")

            # Detect policy units
            policy_units = self._detect_policy_units(text)
            policy_graph["policy_units"].extend(policy_units)

            # Create section
            section = {
                "text": text,
                "page": page.get("page_num"),
                "title": self._extract_title(text),
                "area": None,
                "eje": None,
            }
            policy_graph["sections"].append(section)

        # Extract axes, programs, projects
        for unit in policy_graph["policy_units"]:
            if unit["type"] == "eje":
                policy_graph["axes"].append(unit["name"])
            elif unit["type"] == "programa":
                policy_graph["programs"].append(unit["name"])
            elif unit["type"] == "proyecto":
                policy_graph["projects"].append(unit["name"])

        return policy_graph

    @calibrated_method("saaaaaa.processing.spc_ingestion.structural.StructuralNormalizer._detect_policy_units")
    def _detect_policy_units(self, text: str) -> list[dict[str, Any]]:
        """Detect policy units in text."""
        units = []

        # Simple keyword-based detection
        keywords = {
            "eje": ["eje", "pilar"],
            "programa": ["programa"],

```

```

        "proyecto": ["proyecto"],
        "meta": ["meta"],
        "indicador": ["indicador"],
    }

    for unit_type, keywords_list in keywords.items():
        for keyword in keywords_list:
            if keyword.lower() in text.lower():
                units.append({
                    "type": unit_type,
                    "name": f"{{keyword.capitalize()}} detected",
                })

    return units

@calibrated_method("saaaaaa.processing.spc_ingestion.structural.StructuralNormalizer._
extract_title")
def _extract_title(self, text: str) -> str:
    """Extract title from text."""
    # Simple: first line or first N characters
    lines = text.split("\n")
    if lines:
        return lines[0][:100]
    return ""

===== FILE: src/saaaaaa/scoring/__init__.py =====
"""Scoring module package.

This package provides backward compatibility for code that imports
scoring classes from saaaaaa.scoring.
"""

from saaaaaa.analysis.scoring.scoring import (
    EvidenceStructureError,
    ModalityConfig,
    ModalityValidationError,
    QualityLevel,
    ScoredResult,
    ScoringError,
    ScoringModality,
    ScoringValidator,
    apply_rounding,
    apply_scoring,
    clamp,
    determine_quality_level,
    score_type_a,
    score_type_b,
    score_type_c,
    score_type_d,
    score_type_e,
    score_type_f,
)

__all__ = [
    "EvidenceStructureError",
    "ModalityConfig",
    "ModalityValidationError",
    "QualityLevel",
    "ScoredResult",
    "ScoringError",
    "ScoringModality",
    "ScoringValidator",
    "apply_rounding",
    "apply_scoring",
    "clamp",
    "determine_quality_level",
    "score_type_a",
    "score_type_b",

```

```
"score_type_c",
"score_type_d",
"score_type_e",
"score_type_f",
]
```

```
===== FILE: src/saaaaaa/scoring/scoring.py =====
```

```
"""Scoring module - re-exports from analysis.scoring.scoring.
```

```
This module provides backward compatibility for code that imports
scoring classes from saaaaaa.scoring.scoring.
"""
```

```
from saaaaaa.analysis.scoring.scoring import (
    EvidenceStructureError,
    ModalityConfig,
    ModalityValidationError,
    QualityLevel,
    ScoredResult,
    ScoringError,
    ScoringModality,
    ScoringValidator,
    apply_rounding,
    apply_scoring,
    clamp,
    determine_quality_level,
    score_type_a,
    score_type_b,
    score_type_c,
    score_type_d,
    score_type_e,
    score_type_f,
)
```

```
__all__ = [
    "EvidenceStructureError",
    "ModalityConfig",
    "ModalityValidationError",
    "QualityLevel",
    "ScoredResult",
    "ScoringError",
    "ScoringModality",
    "ScoringValidator",
    "apply_rounding",
    "apply_scoring",
    "clamp",
    "determine_quality_level",
    "score_type_a",
    "score_type_b",
    "score_type_c",
    "score_type_d",
    "score_type_e",
    "score_type_f",
]
```

```
===== FILE: src/saaaaaa/scripts/__init__.py =====
```

```
"""
```

```
Runtime-facing console entrypoints for the SAAAAAA package.
```

```
Modules in this package are designed to be executed via
``python -m saaaaaa.scripts.<name>`` so that repository
scripts no longer need to mutate ``sys.path`` at runtime.
"""
```

```
from __future__ import annotations
```

```
__all__: list[str] = []
```

```
===== FILE: src/saaaaaa/scripts/run_policy_pipeline_verified.py =====
```

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

```
"""
```

```
F.A.R.F.A.N Verified Pipeline Runner
```

```
=====
```

Framework for Advanced Retrieval of Administrativa Narratives

Canonical entrypoint for executing the F.A.R.F.A.N policy analysis pipeline with cryptographic verification and structured claim logging. This script is designed to be machine-auditable and produces verifiable artifacts at every step.

Key Features:

- Computes SHA256 hashes of all inputs and outputs
- Emits structured JSON claims for all operations
- Generates verification\_manifest.json with success status
- Enforces zero-trust validation principles
- No fabricated logs or unverifiable banners

Usage:

```
python -m saaaaaa/scripts/run_policy_pipeline_verified [--plan PLAN_PDF]
```

Requirements:

- Input PDF must exist (default: data/plans/Plan\_1.pdf)
- Package installed via ``pip install -e .``
- Write access to artifacts/ directory

```
"""
```

```
from __future__ import annotations
```

```
import asyncio
```

```
import hashlib
```

```
import json
```

```
import os
```

```
import platform
```

```
import random
```

```
import sys
```

```
import time
```

```
import traceback
```

```
from dataclasses import asdict, dataclass
```

```
from datetime import datetime
```

```
from pathlib import Path
```

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

```
import saaaaaa
```

```
from saaaaaa.config.paths import PROJECT_ROOT
```

```
if os.environ.get("PIPELINE_DEBUG"):
```

```
    print(f"DEBUG: saaaaaa loaded from {saaaaaa.__file__}", flush=True)
```

```
    print(f"DEBUG: sys.path = {sys.path}", flush=True)
```

```
_expected_saaaaaa_prefix = (PROJECT_ROOT / "src" / "saaaaaa").resolve()
```

```
if not Path(saaaaaa.__file__).resolve().is_relative_to(_expected_saaaaaa_prefix):
```

```
    raise RuntimeError(
```

```
        "MODULE SHADOWING DETECTED!\n"
```

```
        f" Expected saaaaaa from: {_expected_saaaaaa_prefix}\n"
```

```
        f" Actually loaded from: {saaaaaa.__file__}\n"
```

```
        "Fix: uninstall old package before running the verified pipeline."
```

```
)
```

```
# Import contract enforcement infrastructure
```

```
from saaaaaa.core.runtime_config import RuntimeConfig, get_runtime_config
```

```
from saaaaaa.core.boot_checks import run_boot_checks, get_boot_check_summary,
```

```
BootCheckError
```

```
from saaaaaa.core.observability.structured_logging import log_runtime_config_loaded
```

```
from saaaaaa.core.orchestrator.seed_registry import get_global_seed_registry
```

```
from saaaaaa.core.orchestrator.verification_manifest import (
```

```
    VerificationManifest as VerificationManifestBuilder,
```

```

        verify_manifest_integrity
    )
from saaaaaa.core.phases.phase2_types import validate_phase2_result
from saaaaaa.core.orchestrator.versions import get_all_versions

@dataclass
class ExecutionClaim:
    """Structured claim about a pipeline operation."""
    timestamp: str
    claim_type: str # "start", "complete", "error", "artifact", "hash"
    component: str
    message: str
    data: Optional[Dict[str, Any]] = None

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

@dataclass
class VerificationManifest:
    """Complete verification manifest for pipeline execution."""
    success: bool
    execution_id: str
    start_time: str
    end_time: str
    input_pdf_path: str
    input_pdf_sha256: str
    artifacts_generated: List[str]
    artifact_hashes: Dict[str, str]
    phases_completed: int
    phases_failed: int
    total_claims: int
    errors: List[str]

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

class VerifiedPipelineRunner:
    """Executes pipeline with cryptographic verification and claim logging."""

    def __init__(self, plan_pdf_path: Path, artifacts_dir: Path, questionnaire_path:
Optional[Path] = None):
        """
        Initialize verified runner.

        Args:
            plan_pdf_path: Path to input PDF
            artifacts_dir: Directory for output artifacts
            questionnaire_path: Optional path to questionnaire file.
                               If None, uses canonical path from
                               saaaaaa.config.paths.QUESTIONNAIRE_FILE
        """
        self.plan_pdf_path = plan_pdf_path
        self.artifacts_dir = artifacts_dir
        self.claims: List[ExecutionClaim] = []
        self.execution_id = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
        self.start_time = datetime.utcnow().isoformat()
        self.phases_completed = 0
        self.phases_failed = 0
        self.errors: List[str] = []
        self.policy_unit_id = f"policy_unit::{self.plan_pdf_path.stem}"
        self.correlation_id = self.execution_id
        self.versions = get_all_versions()
        self.phase2_report: dict[str, Any] | None = None

```

```

self._last_manifest_success: bool = False

# Set questionnaire path (explicit input, SIN_CARRETA compliance)
if questionnaire_path is None:
    from saaaaa.config.paths import QUESTIONNAIRE_FILE
    questionnaire_path = QUESTIONNAIRE_FILE

self.questionnaire_path = questionnaire_path

# Initialize seed registry for deterministic execution
self.seed_registry = get_global_seed_registry()
self.seed_snapshot = self._initialize_determinism_context()

# Initialize verification manifest builder
manifest_secret = (
    os.getenv("VERIFICATION_HMAC_SECRET")
    or os.getenv("MANIFEST_SECRET_KEY")
)
self.manifest_builder = VerificationManifestBuilder(hmac_secret=manifest_secret)
self.manifest_builder.manifest_data["versions"] = dict(self.versions)

# Ensure artifacts directory exists
self.artifacts_dir.mkdir(parents=True, exist_ok=True)

# Initialize runtime configuration
self.runtime_config = RuntimeConfig.from_env()
self.log_claim("start", "runtime_config",
    f"Runtime configuration loaded: {self.runtime_config}",
    {"mode": self.runtime_config.mode.value,
     "strict_mode": self.runtime_config.is_strict_mode()})

# Log runtime config for observability
log_runtime_config_loaded(
    config_repr=repr(self.runtime_config),
    runtime_mode=self.runtime_config.mode
)

def _initialize_determinism_context(self) -> dict[str, int]:
    """
    Seed all deterministic sources (python, numpy, etc.) via SeedRegistry.

    Returns:
        Snapshot of generated seeds keyed by component.
    """
    seeds = self.seed_registry.get_seeds_for_context(
        policy_unit_id=self.policy_unit_id,
        correlation_id=self.correlation_id,
    )

    python_seed = seeds.get("python")
    if python_seed is not None:
        random.seed(python_seed)

    numpy_seed = seeds.get("numpy")
    if numpy_seed is not None:
        try:
            import numpy as np

            np.random.seed(numpy_seed)
        except Exception as exc:
            self.log_claim(
                "warning",
                "determinism",
                f"Failed to seed NumPy RNG: {exc}",
                {"seed": numpy_seed},
            )

    return seeds

```

```

def log_claim(self, claim_type: str, component: str, message: str,
              data: Optional[Dict[str, Any]] = None) -> None:
    """
    Log a structured claim.

    Args:
        claim_type: Type of claim (start, complete, error, artifact, hash)
        component: Component making the claim
        message: Human-readable message
        data: Optional structured data
    """
    claim = ExecutionClaim(
        timestamp=datetime.utcnow().isoformat(),
        claim_type=claim_type,
        component=component,
        message=message,
        data=data or {}
    )
    self.claims.append(claim)

    # Also print for real-time monitoring
    claim_json = json.dumps(claim.to_dict(), separators=(',', ':'))
    print(f"CLAIM: {claim_json}", flush=True)

def compute_sha256(self, file_path: Path) -> str:
    """
    Compute SHA256 hash of a file.

    Args:
        file_path: Path to file

    Returns:
        Hex-encoded SHA256 hash
    """
    sha256_hash = hashlib.sha256()
    with open(file_path, "rb") as f:
        for byte_block in iter(lambda: f.read(4096), b''):
            sha256_hash.update(byte_block)
    return sha256_hash.hexdigest()

def _verify_and_hash_file(self, file_path: Path, file_type: str, attr_name: str) -> bool:
    """
    Verify file exists and compute its SHA256 hash.

    Args:
        file_path: Path to file to verify and hash
        file_type: Human-readable file type (e.g., "Input PDF", "Questionnaire")
        attr_name: Attribute name to store hash (e.g., "input_pdf_sha256")

    Returns:
        True if verification successful, False otherwise
    """
    # Verify file exists
    if not file_path.exists():
        error_msg = f"{file_type} not found: {file_path}"
        self.log_claim("error", "input_verification", error_msg)
        self.errors.append(error_msg)
        return False

    # Compute hash
    try:
        file_hash = self.compute_sha256(file_path)
        setattr(self, attr_name, file_hash)
        self.log_claim("hash", "input_verification",
                       f"{file_type} SHA256: {file_hash}",
                       {"file": str(file_path), "hash": file_hash})
    
```

```

        return True
    except Exception as e:
        error_msg = f"Failed to hash {file_type}: {str(e)}"
        self.log_claim("error", "input_verification", error_msg)
        self.errors.append(error_msg)
        return False

def verify_input(self) -> bool:
    """
    Verify input PDF and questionnaire exist and compute hashes.

    Returns:
        True if all inputs are valid
    """
    self.log_claim("start", "input_verification", "Verifying input files (PDF + questionnaire)")

    # Verify and hash PDF
    if not self._verify_and_hash_file(self.plan_pdf_path, "Input PDF",
    "input_pdf_sha256"):
        return False

    # Verify and hash questionnaire (CRITICAL for SIN_CARRETA compliance)
    if not self._verify_and_hash_file(self.questionnaire_path, "Questionnaire",
    "questionnaire_sha256"):
        return False

    self.log_claim("complete", "input_verification",
        "Input verification successful (PDF + questionnaire)",
        {"pdf_path": str(self.plan_pdf_path),
        "questionnaire_path": str(self.questionnaire_path)})
    return True

def run_boot_checks(self) -> bool:
    """
    Run boot-time validation checks.

    Returns:
        True if all checks pass or fallbacks are allowed

    Raises:
        BootCheckError: If critical check fails in PROD mode
    """
    self.log_claim("start", "boot_checks", "Running boot-time validation checks")

    try:
        results = run_boot_checks(self.runtime_config)
        summary = get_boot_check_summary(results)

        # Log summary
        self.log_claim("complete", "boot_checks",
            f"Boot checks completed\n{summary}",
            {"results": results})

        # Print summary for visibility
        print("\n" + summary + "\n", flush=True)

        return True

    except BootCheckError as e:
        # Critical boot check failed in PROD mode
        error_msg = f"Boot check failed: {e}"
        self.log_claim("error", "boot_checks", error_msg,
            {"component": e.component,
            "code": e.code,
            "reason": e.reason})
        self.errors.append(error_msg)

```

```

# In PROD mode, abort execution
if self.runtime_config.mode.value == "prod":
    print(f"\n ✖ FATAL: {error_msg}\n", flush=True)
    raise

# In DEV/EXPLORATORY, log warning but continue
print(f"\n ⚠ WARNING: {error_msg} (continuing in
{self.runtime_config.mode.value} mode)\n", flush=True)
return False

async def run_spc_ingestion(self) -> Optional[Any]:
    """
    Run SPC (Smart Policy Chunks) ingestion phase - canonical phase-one.

    Passes explicit questionnaire_path to SPC pipeline for SIN_CARRETA compliance.

    Returns:
        SPC object if successful, None otherwise
    """
    self.log_claim("start", "spc_ingestion",
        "Starting SPC ingestion (phase-one) with questionnaire",
        {"questionnaire_path": str(self.questionnaire_path)})

    try:
        from saaaaa.processing.spc_ingestion import CPPIngestionPipeline

        # Pass questionnaire_path explicitly (SIN_CARRETA: no hidden inputs)
        pipeline = CPPIngestionPipeline(questionnaire_path=self.questionnaire_path)
        cpp = await pipeline.process(self.plan_pdf_path)

        self.phases_completed += 1
        self.log_claim("complete", "spc_ingestion",
            "SPC ingestion (phase-one) completed successfully",
            {"phases_completed": self.phases_completed,
             "questionnaire_path": str(self.questionnaire_path)})
        return cpp

    except Exception as e:
        self.phases_failed += 1
        error_msg = f"SPC ingestion failed: {str(e)}"
        self.log_claim("error", "spc_ingestion", error_msg,
            {"traceback": traceback.format_exc()})
        self.errors.append(error_msg)
        return None

async def run_cpp_adapter(self, cpp: Any) -> Optional[Any]:
    """
    Run SPC adapter to convert to PreprocessedDocument.

    Args:
        cpp: CPP/SPC object from ingestion

    Returns:
        PreprocessedDocument if successful, None otherwise
    """
    self.log_claim("start", "spc_adapter", "Starting SPC adaptation")

    try:
        from saaaaa.utils.spc_adapter import SPCAdapter

        # Derive document_id from CPP metadata or fallback to plan filename
        document_id = None
        if hasattr(cpp, "metadata") and isinstance(cpp.metadata, dict):
            document_id = cpp.metadata.get("document_id")
        if not document_id:
            document_id = self.plan_pdf_path.stem

        adapter = SPCAdapter()

```

```

# Pass document_id as required by SPCAdapter API
preprocessed = adapter.to_preprocessed_document(cpp, document_id=document_id)

self.phases_completed += 1
self.log_claim("complete", "spc_adapter",
               "SPC adaptation completed successfully",
               {"phases_completed": self.phases_completed})
return preprocessed

except Exception as e:
    self.phases_failed += 1
    error_msg = f"SPC adaptation failed: {str(e)}"
    self.log_claim("error", "spc_adapter", error_msg,
                  {"traceback": traceback.format_exc()})
    self.errors.append(error_msg)
    return None

async def run_orchestrator(self, preprocessed_doc: Any) -> Optional[list[Any]]:
    """
    Run orchestrator with all phases and verify Phase 2 success.

    Args:
        preprocessed_doc: PreprocessedDocument

    Returns:
        List of PhaseResult objects if successful, None otherwise
    """
    self.log_claim("start", "orchestrator", "Starting orchestrator execution")

    try:
        # This is not the PhaseOrchestrator from the other file, but the core one.
        from saaaaaa.core.orchestrator.factory import build_processor

        processor = build_processor()

        # The core orchestrator is at processor.orchestrator
        results = await processor.orchestrator.process_development_plan_async(
            pdf_path=str(self.plan_pdf_path),
            preprocessed_document=preprocessed_doc
        )

        if not results:
            raise RuntimeError("Orchestrator returned no results.")

        # JOBFROnt 3: Verify Phase 2 (Microquestions) success
        phase2_ok = False
        phase2_report = {"success": False, "question_count": 0, "errors": []}
        if len(results) >= 3:
            phase2_result = results[2] # This is a PhaseResult dataclass
            if phase2_result.success:
                is_valid, validation_errors, normalized_questions =
validate_phase2_result(
                    phase2_result.data
                )
                if is_valid:
                    phase2_ok = True
                    phase2_report["success"] = True
                    phase2_report["question_count"] = len(normalized_questions or [])
                else:
                    error_msg = "Orchestrator Phase 2 failed structural invariant:
questions list is empty or missing."
                    phase2_report["errors"].extend(validation_errors or [])
                    phase2_report["errors"].append(error_msg)
                    self.log_claim("error", "orchestrator", error_msg, {"phase_id":
phase2_result.phase_id})
                    self.errors.append(error_msg)
                else:
                    error_msg = f"Orchestrator Phase 2 failed internally:

```

```

{phase2_result.error}"
    phase2_report["errors"].append(error_msg)
    self.log_claim("error", "orchestrator", error_msg, {"phase_id":
phase2_result.phase_id})
    self.errors.append(error_msg)
else:
    error_msg = "Orchestrator did not produce a result for Phase 2."
    phase2_report["errors"].append(error_msg)
    self.log_claim("error", "orchestrator", error_msg)
    self.errors.append(error_msg)

self.phase2_report = phase2_report

if not phase2_ok:
    # Signal failure as per this script's convention
    self.phases_failed += 1
    return None

# Correctly count completed phases from the results list
completed_phases = sum(1 for r in results if r.success)
self.phases_completed += completed_phases

self.log_claim("complete", "orchestrator",
    "Orchestrator execution completed successfully",
    {"phases_completed": self.phases_completed,
    "core_phases_run": len(results)})
return results

except Exception as e:
    self.phases_failed += 1
    error_msg = f"Orchestrator execution failed: {str(e)}"
    self.log_claim("error", "orchestrator", error_msg,
        {"traceback": traceback.format_exc()})
    self.errors.append(error_msg)
    if self.phase2_report is None:
        self.phase2_report = {"success": False, "question_count": 0, "errors":
[error_msg]}
    return None

def save_artifacts(self, cpp: Any, preprocessed_doc: Any,
    results: Any) -> tuple[List[str], Dict[str, str]]:
    """
    Save artifacts and compute hashes.

    Args:
        cpp: CPP object
        preprocessed_doc: PreprocessedDocument
        results: Orchestrator results

    Returns:
        List of artifact file paths
    """
    self.log_claim("start", "artifact_generation", "Saving artifacts")

    artifacts = []
    artifact_hashes = {}

    try:
        # Save complete CanonPolicyPackage if available (HOSTILE AUDIT REQUIREMENT)
        if cpp:
            cpp_path = self.artifacts_dir / "cpp.json"
            try:
                # Serialize CPP with custom JSON encoder for dataclasses
                from dataclasses import asdict, is_dataclass
                import numpy as np

                def cpp_to_dict(obj):
                    """Convert dataclass/numpy to JSON-serializable format"""

```

```

        if is_dataclass(obj):
            return asdict(obj)
        elif isinstance(obj, np.ndarray):
            return obj.tolist()
        elif isinstance(obj, (np.int64, np.int32)):
            return int(obj)
        elif isinstance(obj, (np.float64, np.float32)):
            return float(obj)
        else:
            return str(obj)

    cpp_dict = asdict(cpp) if is_dataclass(cpp) else {}

    with open(cpp_path, 'w') as f:
        json.dump(cpp_dict, f, indent=2, default=cpp_to_dict)

    artifacts.append(str(cpp_path))
    artifact_hashes[str(cpp_path)] = self.compute_sha256(cpp_path)

    self.log_claim("artifact", "cpp_serialization",
                  f"Serialized complete CanonPolicyPackage",
                  {"file": str(cpp_path),
                   "size_bytes": cpp_path.stat().st_size})

except Exception as e:
    self.log_claim("error", "artifact_generation",
                  f"Failed to serialize CPP: {str(e)}")

# Save preprocessed document metadata
if preprocessed_doc:
    doc_metadata_path = self.artifacts_dir / "preprocessed_doc_metadata.json"
    try:
        with open(doc_metadata_path, 'w') as f:
            json.dump({
                "execution_id": self.execution_id,
                "doc_generated": True,
                "timestamp": datetime.utcnow().isoformat()
            }, f, indent=2)
        artifacts.append(str(doc_metadata_path))
        artifact_hashes[str(doc_metadata_path)] =
self.compute_sha256(doc_metadata_path)
    except Exception as e:
        self.log_claim("error", "artifact_generation",
                      f"Failed to save doc metadata: {str(e)}")

# Save results summary
if results:
    results_path = self.artifacts_dir / "results_summary.json"
    try:
        with open(results_path, 'w') as f:
            json.dump({
                "execution_id": self.execution_id,
                "results_generated": True,
                "timestamp": datetime.utcnow().isoformat()
            }, f, indent=2)
        artifacts.append(str(results_path))
        artifact_hashes[str(results_path)] = self.compute_sha256(results_path)
    except Exception as e:
        self.log_claim("error", "artifact_generation",
                      f"Failed to save results: {str(e)}")

# Save all claims
claims_path = self.artifacts_dir / "execution_claims.json"
with open(claims_path, 'w') as f:
    json.dump([claim.to_dict() for claim in self.claims], f, indent=2)
artifacts.append(str(claims_path))
artifact_hashes[str(claims_path)] = self.compute_sha256(claims_path)

```

```

        self.log_claim("complete", "artifact_generation",
            f"Saved {len(artifacts)} artifacts",
            {"artifact_count": len(artifacts)})

    return artifacts, artifact_hashes

except Exception as e:
    error_msg = f"Failed to save artifacts: {str(e)}"
    self.log_claim("error", "artifact_generation", error_msg)
    self.errors.append(error_msg)
    return artifacts, artifact_hashes

def _collect_calibration_manifest_data(self) -> Dict[str, Any]:
    """Collect calibration metadata for manifest inclusion."""
    calibration_file = PROJECT_ROOT / "config" / "intrinsic_calibration.json"
    if not calibration_file.exists():
        return {}

    try:
        with open(calibration_file, encoding="utf-8") as handle:
            calibration_payload = json.load(handle)

        calibration_hash = hashlib.sha256(
            json.dumps(calibration_payload, sort_keys=True).encode("utf-8")
        ).hexdigest()

        return {
            "version": self.versions.get("calibration"),
            "hash": calibration_hash[:16],
            "methods_calibrated": len(calibration_payload),
            "methods_missing": [],
        }
    except Exception as exc:
        self.log_claim(
            "warning",
            "calibration_manifest",
            f"Unable to read calibration data: {exc}",
            {"path": str(calibration_file)},
        )
        return {}

def _calculate_chunk_metrics(self, preprocessed_doc: Any, results: Any) -> Dict[str, Any]:
    """
    Calculate SPC utilization metrics for verification manifest.

    Args:
        preprocessed_doc: PreprocessedDocument with chunk information
        results: Orchestrator execution results

    Returns:
        Dictionary with chunk metrics
    """
    if preprocessed_doc is None:
        return {}

    processing_mode = getattr(preprocessed_doc, 'processing_mode', 'flat')

    if processing_mode != 'chunked':
        return {
            "processing_mode": "flat",
            "note": "Document processed in flat mode (no chunk utilization)"
        }

    chunks = getattr(preprocessed_doc, 'chunks', [])
    chunk_graph = getattr(preprocessed_doc, 'chunk_graph', {})

    chunk_metrics = {

```

```

    "processing_mode": "chunked",
    "total_chunks": len(chunks),
    "chunk_types": {},
    "chunk_routing": {},
    "graph_metrics": {},
    "execution_savings": {}
}

# Count chunk types
for chunk in chunks:
    chunk_type = getattr(chunk, 'chunk_type', 'unknown')
    chunk_metrics["chunk_types"][chunk_type] = \
        chunk_metrics["chunk_types"].get(chunk_type, 0) + 1

# Calculate graph metrics if networkx available
try:
    import networkx as nx

    if chunk_graph and isinstance(chunk_graph, dict):
        nodes = chunk_graph.get("nodes", [])
        edges = chunk_graph.get("edges", [])

        # Build networkx graph for analysis
        G = nx.DiGraph()
        for node in nodes:
            node_id = node.get("id")
            if node_id is not None:
                G.add_node(node_id)

        for edge in edges:
            source = edge.get("source")
            target = edge.get("target")
            if source is not None and target is not None:
                G.add_edge(source, target)

        chunk_metrics["graph_metrics"] = {
            "nodes": G.number_of_nodes(),
            "edges": G.number_of_edges(),
            "is_dag": nx.is_directed_acyclic_graph(G),
            "is_connected": nx.is_weakly_connected(G) if G.number_of_nodes() > 0
else False,
            "density": round(nx.density(G), 4) if G.number_of_nodes() > 0 else
0.0,
        }

        # Calculate diameter if connected
        if chunk_metrics["graph_metrics"]["is_connected"]:
            try:
                chunk_metrics["graph_metrics"]["diameter"] =
nx.diameter(G.to_undirected())
            except Exception:
                chunk_metrics["graph_metrics"]["diameter"] = -1
            else:
                chunk_metrics["graph_metrics"]["diameter"] = -1

except ImportError:
    chunk_metrics["graph_metrics"] = {
        "note": "NetworkX not available for graph analysis"
    }
except Exception as e:
    chunk_metrics["graph_metrics"] = {
        "error": f"Graph analysis failed: {str(e)}"
    }

# Calculate execution savings
# Use actual metrics from orchestrator if available
if results and hasattr(results, '_execution_metrics') and 'phase_2' in
results._execution_metrics:

```

```

metrics = results._execution_metrics['phase_2']
chunk_metrics["execution_savings"] = {
    "chunk_executions": metrics['chunk_executions'],
    "full_doc_executions": metrics['full_doc_executions'],
    "total_possible_executions": metrics['total_possible_executions'],
    "actual_executions": metrics['actual_executions'],
    "savings_percent": round(metrics['savings_percent'], 2),
    "note": "Actual execution counts from orchestrator Phase 2"
}
elif results:
    # Fallback to estimation if real metrics not available
    total_possible_executions = 30 * len(chunks) # 30 executors per chunk max
    # Assume chunk routing reduces executions by using type-specific executors
    estimated_actual = len(chunks) * 10 # ~10 executors per chunk (conservative)

    chunk_metrics["execution_savings"] = {
        "total_possible_executions": total_possible_executions,
        "estimated_actual_executions": estimated_actual,
        "estimated_savings_percent": round(
            (1 - estimated_actual / max(total_possible_executions, 1)) * 100, 2
        ) if total_possible_executions > 0 else 0.0,
        "note": "Estimated savings based on chunk-aware routing (orchestrator
metrics not available)"
    }

return chunk_metrics

def _calculate_signal_metrics(self, results: Any) -> Dict[str, Any]:
    """
    Calculate signal utilization metrics for verification manifest.

    Args:
        results: Orchestrator execution results

    Returns:
        Dictionary with signal metrics
    """
    # Try to extract signal usage from results
    try:
        signal_metrics = {
            "enabled": True,
            "transport": "memory",
            "policy_areas_loaded": 10,
        }

        # Check if results have executor information
        if results and hasattr(results, 'executor_metadata'):
            # Count executors that used signals
            executors_with_signals = 0
            total_executors = 0

            for metadata in results.executor_metadata.values():
                total_executors += 1
                if metadata.get('signal_usage'):
                    executors_with_signals += 1

            signal_metrics["executors_using_signals"] = executors_with_signals
            signal_metrics["total_executors"] = total_executors

        # Default values if we can't extract from results
        if "executors_using_signals" not in signal_metrics:
            signal_metrics["executors_using_signals"] = 0
            signal_metrics["total_executors"] = 0
            signal_metrics["note"] = "Signal infrastructure initialized, actual usage
not tracked in results"

        # Add signal pack versions
        signal_metrics["signal_versions"] = {

```

```

        f"PA{i:02d}": "1.0.0" for i in range(1, 11)
    }

    return signal_metrics

except Exception as e:
    # If signal system not initialized, return minimal info
    return {
        "enabled": False,
        "note": f"Signal system not initialized: {str(e)}"
    }

def generate_verification_manifest(
    self,
    artifacts: List[str],
    artifact_hashes: Dict[str, str],
    preprocessed_doc: Any = None,
    results: Any = None
) -> Path:
    """
    Generate final verification manifest with SPC utilization metrics and
    cryptographic integrity.

    Args:
        artifacts: List of artifact paths
        artifact_hashes: Dictionary mapping paths to SHA256 hashes
        preprocessed_doc: PreprocessedDocument (optional, for chunk metrics)
        results: Orchestrator results (optional, for execution metrics)

    Returns:
        Path to verification_manifest.json
    """
    end_time = datetime.utcnow().isoformat()

    # Calculate chunk utilization metrics
    chunk_metrics = self._calculate_chunk_metrics(preprocessed_doc, results)

    # HOSTILE AUDIT: Validate critical invariants before declaring success
    hostile_failures: list[str] = []

    if preprocessed_doc:
        chunk_count = len(getattr(preprocessed_doc, "chunks", []))
        if chunk_count < 5:
            hostile_failures.append(f"chunk_graph too small: {chunk_count} < 5")

    phase2_entry = {
        "name": "Phase 2 – Micro Questions",
        "success": bool(self.phase2_report and self.phase2_report.get("success")),
        "question_count": (self.phase2_report or {}).get("question_count", 0),
        "errors": list((self.phase2_report or {}).get("errors", [])),
    }
    if not phase2_entry["success"] and not phase2_entry["errors"]:
        phase2_entry["errors"].append("Phase 2 not executed")

    # Determine success based on strict criteria + hostile invariants
    success = all(
        [
            self.phases_failed == 0,
            self.phases_completed > 0,
            len(self.errors) == 0,
            len(artifacts) > 0,
            len(hostile_failures) == 0,
            phase2_entry["success"],
        ]
    )

    if hostile_failures:
        self.log_claim(

```

```

        "error",
        "hostile_audit",
        f"Hostile audit failures: {hostile_failures}",
    )
    self.errors.extend(hostile_failures)

builder = self.manifest_builder
builder.manifest_data["versions"] = dict(self.versions)
self._last_manifest_success = success
builder.set_success(success)
builder.set_pipeline_hash(getattr(self, "input_pdf_sha256", ""))
builder.set_environment()

# Determinism metadata
seed_entry = self.seed_registry.get_manifest_entry(
    policy_unit_id=self.policy_unit_id,
    correlation_id=self.correlation_id,
)
builder.set_determinism(
    seed_version=seed_entry.get("seed_version", ""),
    policy_unit_id=seed_entry.get("policy_unit_id"),
    correlation_id=seed_entry.get("correlation_id"),
    seeds_by_component=seed_entry.get("seeds_by_component"),
)

# Calibration metadata
calibration_manifest = self._collect_calibration_manifest_data()
if calibration_manifest:
    builder.set_calibrations(
        calibration_manifest["version"],
        calibration_manifest["hash"],
        calibration_manifest["methods_calibrated"],
        calibration_manifest["methods_missing"],
    )

# Ingestion metadata
if preprocessed_doc:
    raw_text = getattr(preprocessed_doc, "raw_text", "") or ""
    sentences = getattr(preprocessed_doc, "sentences", []) or []
    chunk_count = len(getattr(preprocessed_doc, "chunks", []))
    builder.set_ingestion(
        method="SPC",
        chunk_count=chunk_count,
        text_length=len(raw_text),
        sentence_count=len(sentences),
        chunk_strategy="semantic",
        chunk_overlap=50,
    )

builder.manifest_data.setdefault("phases", {})
builder.manifest_data["phases"]["phase2"] = phase2_entry

# Phase metadata
duration_seconds = (
    datetime.fromisoformat(end_time) - datetime.fromisoformat(self.start_time)
).total_seconds()
builder.add_phase(
    phase_id=0,
    phase_name="complete_pipeline",
    success=success,
    duration_ms=int(duration_seconds * 1000),
    items_processed=self.phases_completed,
    error="; ".join(self.errors) if self.errors and not success else None,
)

# Artifacts
for index, artifact_path in enumerate(sorted(artifact_hashes.keys())):
    artifact_file = Path(artifact_path)

```

```

size_bytes = artifact_file.stat().st_size if artifact_file.exists() else None
builder.add_artifact(
    artifact_id=f"artifact_{index:02d}",
    path=str(artifact_file),
    artifact_hash=artifact_hashes[artifact_path],
    size_bytes=size_bytes,
)

if hasattr(self, "questionnaire_sha256"):
    questionnaire_size = (
        self.questionnaire_path.stat().st_size
        if self.questionnaire_path.exists()
        else None
    )
    builder.add_artifact(
        artifact_id="questionnaire_source",
        path=str(self.questionnaire_path),
        artifact_hash=self.questionnaire_sha256,
        size_bytes=questionnaire_size,
    )
    self.log_claim(
        "artifact",
        "questionnaire",
        "Questionnaire added to manifest",
        {
            "path": str(self.questionnaire_path),
            "hash": self.questionnaire_sha256,
        },
    )

if chunk_metrics:
    builder.manifest_data["spc_utilization"] = chunk_metrics

signal_metrics = self._calculate_signal_metrics(results)
if signal_metrics:
    builder.manifest_data["signals"] = signal_metrics

builder.manifest_data.update(
    {
        "execution_id": self.execution_id,
        "start_time": self.start_time,
        "end_time": end_time,
        "input_pdf_path": str(self.plan_pdf_path),
        "total_claims": len(self.claims),
        "errors": list(self.errors),
        "artifacts_generated": list(artifacts),
        "artifact_hashes": dict(artifact_hashes),
    }
)

manifest_path = self.artifacts_dir / "verification_manifest.json"
manifest_dict = builder.build()
manifest_path.write_text(json.dumps(manifest_dict, indent=2), encoding="utf-8")

hmac_secret = builder.hmac_secret
is_valid = True
if hmac_secret:
    is_valid = verify_manifest_integrity(manifest_dict, hmac_secret)
    if is_valid:
        self.log_claim(
            "hash",
            "verification_manifest",
            "Manifest integrity verified",
            {"file": str(manifest_path)},
        )
    else:
        self.log_claim(
            "error",

```

```

        "verification_manifest",
        "Manifest integrity verification failed",
    )
else:
    self.log_claim(
        "warning",
        "verification_manifest",
        "No HMAC secret provided; integrity verification skipped",
    )

if success and is_valid:
    print("\n" + "=" * 80)
    print("PIPELINE_VERIFIED=1")
    print(f"Manifest: {manifest_path}")
    print(f"HMAC: {manifest_dict.get('integrity_hmac', 'N/A')[:16]}...")
    print(
        f"Phases: {self.phases_completed} completed, {self.phases_failed} failed"
    )
    print(f"Artifacts: {len(artifacts)}")
    print("=" * 80 + "\n")

return manifest_path

async def run(self) -> bool:
    """
    Execute the complete verified pipeline.

    Returns:
        True if pipeline succeeded, False otherwise
    """
    self.log_claim("start", "pipeline", "Starting verified pipeline execution")

    # Step 1: Verify input
    if not self.verify_input():
        self.generate_verification_manifest([], {})
        return False

    # Step 1.5: Run boot checks
    try:
        if not self.run_boot_checks():
            # Boot checks failed but we're in DEV mode - log warning
            self.log_claim("warning", "boot_checks",
                "Boot checks failed but continuing in non-PROD mode")
    except BootCheckError:
        # Boot check failed in PROD mode - abort
        self.generate_verification_manifest([], {})
        return False

    # Step 2: Run SPC ingestion (canonical phase-one)
    cpp = await self.run_spc_ingestion()
    if cpp is None:
        self.generate_verification_manifest([], {})
        return False

    # Step 3: Run CPP adapter
    preprocessed_doc = await self.run_cpp_adapter(cpp)
    if preprocessed_doc is None:
        self.generate_verification_manifest([], {})
        return False

    # Step 4: Run orchestrator
    results = await self.run_orchestrator(preprocessed_doc)
    if results is None:
        self.generate_verification_manifest([], {})
        return False

    # Step 5: Save artifacts
    artifacts, artifact_hashes = self.save_artifacts(cpp, preprocessed_doc, results)

```

```

# Step 6: Generate verification manifest with chunk metrics
manifest_path = self.generate_verification_manifest(
    artifacts, artifact_hashes, preprocessed_doc, results
)

self.log_claim("complete", "pipeline",
    "Pipeline execution completed",
    {
        "success": self._last_manifest_success,
        "phases_completed": self.phases_completed,
        "phases_failed": self.phases_failed,
        "manifest_path": str(manifest_path)
    })

return bool(self._last_manifest_success)

async def main():
    """Main entry point."""
    import argparse

    parser = argparse.ArgumentParser(
        description="Run verified policy pipeline with cryptographic verification"
    )
    parser.add_argument(
        "--plan",
        type=str,
        default="data/plans/Plan_1.pdf",
        help="Path to plan PDF (default: data/plans/Plan_1.pdf)"
    )
    parser.add_argument(
        "--artifacts-dir",
        type=str,
        default="artifacts/plan1",
        help="Directory for artifacts (default: artifacts/plan1)"
    )

    args = parser.parse_args()

    # Resolve paths
    plan_path = PROJECT_ROOT / args.plan
    artifacts_dir = PROJECT_ROOT / args.artifacts_dir

    print("=" * 80, flush=True)
    print("F.A.R.F.A.N VERIFIED POLICY PIPELINE RUNNER", flush=True)
    print("Framework for Advanced Retrieval of Administrativa Narratives", flush=True)
    print("=" * 80, flush=True)
    print(f"Plan: {plan_path}", flush=True)
    print(f"Artifacts: {artifacts_dir}", flush=True)
    print("=" * 80, flush=True)

    # Create and run pipeline
    runner = VerifiedPipelineRunner(plan_path, artifacts_dir)
    success = await runner.run()

    print("=" * 80, flush=True)
    if success:
        print("PIPELINE_VERIFIED=1", flush=True)
        print("Status: SUCCESS", flush=True)
    else:
        print("PIPELINE_VERIFIED=0", flush=True)
        print("Status: FAILED", flush=True)
    print("=" * 80, flush=True)

    sys.exit(0 if success else 1)

```

```
def cli() -> None:
    """Synchronous entripoint for console scripts."""
    asyncio.run(main())

if __name__ == "__main__":
    cli()
```

```
===== FILE: src/saaaaaa/utls/__init__.py =====
"""Utility functions and helpers for SAAAAAA system."""
```

```
===== FILE: src/saaaaaa/utls/contract_io.py =====
"""
```

Contract I/O Envelope - Universal Metadata Wrapper  
 =====

ContractEnvelope wraps every phase payload (input and output) with universal metadata including schema\_version, timestamp\_utc, policy\_unit\_id, content\_digest, correlation\_id, and deterministic event\_id.

This module provides the canonical wrapper for all phase I/O to ensure consistent metadata handling across the pipeline.

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

```
from __future__ import annotations
```

```
import hashlib
import json
from datetime import datetime, timezone
from typing import TYPE_CHECKING, Any
```

```
from pydantic import BaseModel, Field, field_validator
from saaaaaa.core.calibration.decorators import calibrated_method
```

```
if TYPE_CHECKING:
    from collections.abc import Mapping
```

```
CANONICAL_SCHEMA_VERSION = "io-1.0"
```

```
def _canonical_json_bytes(obj: Any) -> bytes:
    """
    Convert object to canonical JSON bytes for deterministic hashing.

    Args:
        obj: Object to serialize

    Returns:
        Canonical JSON bytes (sorted keys, compact separators)
    """
    return json.dumps(
        obj,
        separators=(",", ":"),
        sort_keys=True,
        ensure_ascii=False
    ).encode("utf-8")
```

```
def sha256_hex(obj: Any) -> str:
    """
    Compute SHA-256 hex digest of object via canonical JSON.

    Args:
        obj: Object to hash
```

Returns:

64-character hexadecimal SHA-256 digest

Examples:

```
>>> sha256_hex({"b": 2, "a": 1})
'eed6d51ab37ca6df16a330c85094467efcab7b5746c0e02bc728a05069ede38b'
>>> sha256_hex({"a": 1, "b": 2}) # Same despite key order
'eed6d51ab37ca6df16a330c85094467efcab7b5746c0e02bc728a05069ede38b'
"""
return hashlib.sha256(_canonical_json_bytes(obj)).hexdigest()
```

def utcnow\_iso() -> str:

"""

Get current UTC timestamp in ISO-8601 format with Z suffix.

Returns:

ISO-8601 timestamp string with Z suffix

Examples:

```
>>> ts = utcnow_iso()
>>> ts.endswith('Z')
True
>>> 'T' in ts
True
"""
# Always Z-suffixed UTC; forbidden to use local time
return datetime.now(timezone.utc).isoformat().replace("+00:00", "Z")
```

class ContractEnvelope(BaseModel):

"""

Universal metadata wrapper for phase I/O payloads.

Wraps every phase input/output with consistent metadata for:

- Schema versioning
- Temporal tracking (UTC only)
- Policy scope identification
- Cryptographic verification
- Request correlation
- Deterministic event tracking

Fields:

- schema\_version: Logical envelope schema version
- timestamp\_utc: ISO-8601 Z-suffixed timestamp
- policy\_unit\_id: Stable identifier selecting seeds & scope
- correlation\_id: Client-supplied request/run correlation
- content\_digest: SHA-256 over canonical JSON of `payload`
- event\_id: Deterministic SHA-256 over (policy\_unit\_id, content\_digest)
- payload: The phase's typed deliverable/expectation

Examples:

```
>>> payload = {"result": "success", "data": [1, 2, 3]}
>>> env = ContractEnvelope.wrap(
...     payload,
...     policy_unit_id="PDM-001",
...     correlation_id="req-123"
... )
>>> env.policy_unit_id
'PDM-001'
>>> len(env.content_digest)
64
>>> len(env.event_id)
64
"""
```

schema\_version: str = Field(default=CANONICAL\_SCHEMA\_VERSION)

```

timestamp_utc: str = Field(default_factory=utcnow_iso)
policy_unit_id: str = Field(min_length=1)
correlation_id: str | None = Field(default=None)
content_digest: str = Field(min_length=64, max_length=64)
event_id: str = Field(min_length=64, max_length=64)
payload: Mapping[str, Any] | Any

```

```

model_config = {"frozen": True, "extra": "forbid"}

```

```

@field_validator("timestamp_utc")
@classmethod
def _validate_utc(cls, v: str) -> str:
    """Validate timestamp is Z-suffixed UTC ISO-8601."""
    # Quick sanity: must end with Z and parse
    if not v.endswith("Z"):
        raise ValueError("timestamp_utc must be Z-suffixed UTC (ISO-8601)")
    # Let it raise if invalid
    datetime.fromisoformat(v.replace("Z", "+00:00"))
    return v

```

```

@classmethod
def wrap(
    cls,
    payload: Any,
    *,
    policy_unit_id: str,
    correlation_id: str | None = None,
    schema_version: str = CANONICAL_SCHEMA_VERSION,
) -> ContractEnvelope:
    """

```

Wrap a payload with universal metadata envelope.

Args:

payload: The phase deliverable/expectation to wrap  
 policy\_unit\_id: Policy unit identifier  
 correlation\_id: Optional request correlation ID  
 schema\_version: Schema version (default: io-1.0)

Returns:

ContractEnvelope with computed digests and event ID

Examples:

```

>>> payload = {"status": "ok"}
>>> env = ContractEnvelope.wrap(payload, policy_unit_id="PDM-001")
>>> env.payload["status"]
'ok'
>>> env.policy_unit_id
'PDM-001'
"""
digest = sha256_hex(payload)
event_id = sha256_hex({"policy_unit_id": policy_unit_id, "digest": digest})
return cls(
    schema_version=schema_version,
    policy_unit_id=policy_unit_id,
    correlation_id=correlation_id,
    content_digest=digest,
    event_id=event_id,
    payload=payload,
)

```

```

if __name__ == "__main__":
    import doctest

```

```

# Run doctests
print("Running doctests...")
doctest.testmod(verbose=True)

```

```

# Minimal deterministic check
print("\n" + "="*60)
print("ContractEnvelope Integration Tests")
print("="*60)

print("\n1. Testing deterministic digest computation:")
p = {"a": 1, "b": [2, 3]}
e1 = ContractEnvelope.wrap(p, policy_unit_id="PU_123")
e2 = ContractEnvelope.wrap({"b": [2, 3], "a": 1}, policy_unit_id="PU_123")
assert e1.content_digest == e2.content_digest # canonical JSON stable
assert e1.event_id == e2.event_id
print(f"    ✓ Digest is deterministic: {e1.content_digest[:16]}...")
print(f"    ✓ Event ID is deterministic: {e1.event_id[:16]}...")

print("\n2. Testing envelope immutability:")
try:
    e1.payload = {"modified": True}
    print("    ✗ FAILED: Envelope should be immutable")
except Exception:
    print("    ✓ Envelope is immutable (frozen)")

print("\n3. Testing UTC timestamp validation:")
assert e1.timestamp_utc.endswith('Z')
print(f"    ✓ Timestamp is UTC: {e1.timestamp_utc}")

print("\n4. Testing correlation ID:")
e3 = ContractEnvelope.wrap(p, policy_unit_id="PU_123", correlation_id="corr-456")
assert e3.correlation_id == "corr-456"
print(f"    ✓ Correlation ID: {e3.correlation_id}")

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

```

===== FILE: src/saaaaaa/utls/contracts.py =====

"""

CONTRACT DEFINITIONS - Frozen Data Shapes

=====

TypedDict and Protocol definitions for API contracts across modules.  
 All data shapes must be versioned and adapters maintained for one release cycle.

Purpose: Replace ad-hoc dicts with typed structures to prevent:

- unexpected keyword argument errors
- missing required positional arguments
- 'str' object has no attribute 'text' errors
- 'bool' object is not iterable
- unhashable type: 'dict' in sets

Version 2.0 Enhancement:

- Pydantic-based contracts with strict validation (enhanced\_contracts.py)
- Backward compatibility with V1 TypedDict contracts maintained
- Domain-specific exceptions and structured logging
- Cryptographic content verification and deterministic execution

"""

```
from __future__ import annotations
```

```
from dataclasses import dataclass
```

```
from typing import (
```

```
    TYPE_CHECKING,
```

```
    Any,
```

```
    Literal,
```

```
    Protocol,
```

```
    TypedDict,
```

```
)
```

```
if TYPE_CHECKING:
```

```

from collections.abc import Iterable, Mapping, Sequence
from pathlib import Path

# =====
# V2 ENHANCED CONTRACTS - Pydantic-based with strict validation
# =====
# Import V2 contracts from enhanced_contracts module
# Use these for new code; V1 contracts maintained for backward compatibility
from .enhanced_contracts import (
    from saaaaaa.core.calibration.decorators import calibrated_method
    # Pydantic Models
    AnalysisInputV2,
    AnalysisOutputV2,
    BaseContract,
    # Exceptions
    ContractValidationError,
    DataIntegrityError,
    DocumentMetadataV2,
    ExecutionContextV2,
    FlowCompatibilityError,
    ProcessedTextV2,
    # Utilities
    StructuredLogger,
    SystemConfigError,
    compute_content_digest,
    utc_now_iso,
)

# =====
# DOCUMENT CONTRACTS - V1
# =====

class DocumentMetadataV1(TypedDict, total=True):
    """Document metadata shape - all fields required."""
    file_path: str
    file_name: str
    num_pages: int
    file_size_bytes: int
    file_hash: str

class DocumentMetadataV1Optional(TypedDict, total=False):
    """Optional document metadata fields."""
    pdf_metadata: dict[str, Any]
    author: str
    title: str
    creation_date: str

class ProcessedTextV1(TypedDict, total=True):
    """Shape for processed text output."""
    raw_text: str
    normalized_text: str
    language: str
    encoding: str

class ProcessedTextV1Optional(TypedDict, total=False):
    """Optional processed text fields."""
    sentences: list[str]
    sections: list[dict[str, Any]]
    tables: Mapping[str, Any]

# =====
# ANALYSIS CONTRACTS - V1
# =====

class AnalysisInputV1(TypedDict, total=True):
    """Required fields for analysis input - keyword-only."""
    text: str
    document_id: str

```

```

class AnalysisInputV1Optional(TypedDict, total=False):
    """Optional fields for analysis input."""
    metadata: Mapping[str, Any]
    context: Mapping[str, Any]
    sentences: Sequence[str]

class AnalysisOutputV1(TypedDict, total=True):
    """Shape for analysis output."""
    dimension: str
    category: str
    confidence: float
    matches: Sequence[str]

class AnalysisOutputV1Optional(TypedDict, total=False):
    """Optional analysis output fields."""
    positions: Sequence[int]
    evidence: Sequence[str]
    warnings: Sequence[str]

# =====
# EXECUTION CONTRACTS - V1
# =====

class ExecutionContextV1(TypedDict, total=True):
    """Execution context for method invocation."""
    class_name: str
    method_name: str
    document_id: str

class ExecutionContextV1Optional(TypedDict, total=False):
    """Optional execution context fields."""
    raw_text: str
    text: str
    metadata: Mapping[str, Any]
    tables: Mapping[str, Any]
    sentences: Sequence[str]

# =====
# ERROR REPORTING CONTRACTS
# =====

class ContractMismatchError(TypedDict, total=True):
    """Standard error shape for contract mismatches."""
    error_code: Literal["ERR_CONTRACT_MISMATCH"]
    stage: str
    function: str
    parameter: str
    expected_type: str
    got_type: str
    producer: str
    consumer: str

# =====
# PROTOCOLS FOR PLUGGABLE BEHAVIOR
# =====

class TextProcessorProtocol(Protocol):
    """Protocol for text processing components."""

    @calibrated_method("saaaaaa.utils.contracts.TextProcessorProtocol.normalize_unicode")
    def normalize_unicode(self, text: str) -> str:
        """Normalize unicode characters in text."""
        ...

    @calibrated_method("saaaaaa.utils.contracts.TextProcessorProtocol.segment_into_sentences")
    def segment_into_sentences(self, text: str) -> Sequence[str]:

```

```

"""Segment text into sentences."""
...

class DocumentLoaderProtocol(Protocol):
    """Protocol for document loading components."""

    @calibrated_method("saaaaaa.utils.contracts.DocumentLoaderProtocol.load_pdf")
    def load_pdf(self, *, pdf_path: Path) -> DocumentMetadataV1:
        """Load PDF and return metadata - keyword-only params."""
        ...

    @calibrated_method("saaaaaa.utils.contracts.DocumentLoaderProtocol.validate_pdf")
    def validate_pdf(self, *, pdf_path: Path) -> bool:
        """Validate PDF file - keyword-only params."""
        ...

class AnalyzerProtocol(Protocol):
    """Protocol for analysis components."""

    def analyze(
        self,
        *,
        text: str,
        document_id: str,
        metadata: Mapping[str, Any] | None = None,
    ) -> AnalysisOutputV1:
        """Analyze text and return structured output - keyword-only params."""
        ...

# =====
# VALUE OBJECTS (prevent .text on strings)
# =====

@dataclass(frozen=True, slots=True)
class TextDocument:
    """Wrapper to prevent passing plain str where structured text is required."""
    text: str
    document_id: str
    metadata: Mapping[str, Any]

    def __post_init__(self) -> None:
        """Validate that text is non-empty."""
        if not isinstance(self.text, str):
            raise TypeError(
                f"ERR_CONTRACT_MISMATCH: text must be str, got {type(self.text).__name__}"
            )
        if not self.text:
            raise ValueError("ERR_CONTRACT_MISMATCH: text cannot be empty")

@dataclass(frozen=True, slots=True)
class SentenceCollection:
    """Type-safe collection of sentences (prevents iteration bugs)."""
    sentences: tuple[str, ...] # Immutable and hashable

    def __post_init__(self) -> None:
        """Validate sentences are strings."""
        if not all(isinstance(s, str) for s in self.sentences):
            raise TypeError(
                "ERR_CONTRACT_MISMATCH: All sentences must be strings"
            )

    def __iter__(self) -> Iterable[str]:
        """Make iterable."""
        return iter(self.sentences)

    def __len__(self) -> int:
        """Return count."""
        return len(self.sentences)

```

```

# =====
# SENTINEL VALUES (avoid None ambiguity)
# =====

class _MissingSentinel:
    """Sentinel type for missing optional parameters."""

    def __repr__(self) -> str:
        return "<MISSING>"

MISSING: _MissingSentinel = _MissingSentinel()

# =====
# RUNTIME VALIDATION HELPERS
# =====

def validate_contract(
    value: Any,
    expected_type: type,
    *,
    parameter: str,
    producer: str,
    consumer: str,
) -> None:
    """
    Validate value matches expected contract at runtime.

    Raises TypeError with structured error message on mismatch.
    """
    if not isinstance(value, expected_type):
        error_msg = (
            f"ERR_CONTRACT_MISMATCH["
            f"param='{parameter}', "
            f"expected={expected_type.__name__}, "
            f"got={type(value).__name__}, "
            f"producer={producer}, "
            f"consumer={consumer}"
            f"]"
        )
        raise TypeError(error_msg)

def validate_mapping_keys(
    mapping: Mapping[str, Any],
    required_keys: Sequence[str],
    *,
    producer: str,
    consumer: str,
) -> None:
    """
    Validate mapping contains required keys.

    Raises KeyError with structured message on missing keys.
    """
    missing = [key for key in required_keys if key not in mapping]
    if missing:
        error_msg = (
            f"ERR_CONTRACT_MISMATCH["
            f"missing_keys={missing}, "
            f"producer={producer}, "
            f"consumer={consumer}"
            f"]"
        )
        raise KeyError(error_msg)

def ensure_iterable_not_string(
    value: Any,
    *,

```

```

parameter: str,
producer: str,
consumer: str,
) -> None:
"""
Validate value is iterable but NOT a string or bytes.

```

Prevents "'bool' object is not iterable" and "iterate string as tokens" bugs.

```

"""
if isinstance(value, (str, bytes)):
    raise TypeError(
        f"ERR_CONTRACT_MISMATCH["
        f"param='{parameter}', "
        f"expected=Iterable (not str/bytes), "
        f"got={type(value).__name__}, "
        f"producer={producer}, "
        f"consumer={consumer}"
        f"]"
    )

```

```

try:
    iter(value)
except TypeError as e:
    raise TypeError(
        f"ERR_CONTRACT_MISMATCH["
        f"param='{parameter}', "
        f"expected=Iterable, "
        f"got={type(value).__name__}, "
        f"producer={producer}, "
        f"consumer={consumer}"
        f"]"
    ) from e

```

```

def ensure_hashable(
    value: Any,
    *,
    parameter: str,
    producer: str,
    consumer: str,
) -> None:
"""
Validate value is hashable (can be added to set or used as dict key).

```

Prevents "unhashable type: 'dict'" errors.

```

"""
try:
    hash(value)
except TypeError as e:
    raise TypeError(
        f"ERR_CONTRACT_MISMATCH["
        f"param='{parameter}', "
        f"expected=Hashable, "
        f"got={type(value).__name__} (unhashable), "
        f"producer={producer}, "
        f"consumer={consumer}"
        f"]"
    ) from e

```

```

# =====
# MODULE EXPORTS
# =====

```

```

__all__ = [
    # V2 Enhanced Contracts (Pydantic-based) - RECOMMENDED FOR NEW CODE
    "AnalysisInputV2",
    "AnalysisOutputV2",
    "BaseContract",

```

```

"DocumentMetadataV2",
"ExecutionContextV2",
"ProcessedTextV2",
# V2 Exceptions
"ContractValidationError",
"DataIntegrityError",
"FlowCompatibilityError",
"SystemConfigError",
# V2 Utilities
"StructuredLogger",
"compute_content_digest",
"utc_now_iso",
# V1 Contracts (TypedDict-based) - BACKWARD COMPATIBILITY
"AnalysisInputV1",
"AnalysisInputV1Optional",
"AnalysisOutputV1",
"AnalysisOutputV1Optional",
"AnalyzerProtocol",
"ContractMismatchError",
"DocumentLoaderProtocol",
"DocumentMetadataV1",
"DocumentMetadataV1Optional",
"ExecutionContextV1",
"ExecutionContextV1Optional",
"MISSING",
"ProcessedTextV1",
"ProcessedTextV1Optional",
"SentenceCollection",
"TextDocument",
"TextProcessorProtocol",
"ensure_hashable",
"ensure_iterable_not_string",
"validate_contract",
"validate_mapping_keys",
]

```

```

===== FILE: src/saaaaaa/utls/contracts_runtime.py =====
"""

```

Runtime Contract Validation using Pydantic.

This module provides runtime validators for all TypedDict contracts defined in core\_contracts.py. These validators enforce:

- Value bounds and constraints
- Required vs optional fields with strict validation
- Schema versioning for backward compatibility
- Round-trip serialization guarantees

The validators mirror the TypedDict shapes exactly but add runtime enforcement. Use these at public API boundaries and orchestrator edges.

```

Version: 1.0.0
Schema Version Format: sem-{major}.{minor}
"""

```

```

from typing import Any

```

```

from pydantic import BaseModel, ConfigDict, Field, field_validator
from saaaaaa.core.calibration.decorators import calibrated_method

```

```

# =====
# CONFIGURATION
# =====

```

```

class StrictModel(BaseModel):
    """Base model with strict configuration for all contract validators."""

    model_config = ConfigDict(
        extra='forbid', # Refuse unknown fields
    )

```

```

        validate_assignment=True,
        str_strip_whitespace=True,
        validate_default=True,
        populate_by_name=True, # Allow both field name and alias
    )

# =====
# ANALYZER_ONE.PY CONTRACTS
# =====

class SemanticAnalyzerInputModel(StrictModel):
    """Runtime validator for SemanticAnalyzerInputContract.

    Validates:
    - text is non-empty
    - schema_version follows sem-X.Y pattern
    - segments is a list of strings
    - ontology_params is a valid dict

    Example:
    >>> model = SemanticAnalyzerInputModel(
    ...     text="El plan de desarrollo municipal...",
    ...     schema_version="sem-1.0"
    ... )
    """
    text: str = Field(min_length=1, description="Document text to analyze")
    segments: list[str] = Field(
        default_factory=list,
        description="Pre-segmented text chunks"
    )
    ontology_params: dict[str, Any] = Field(
        default_factory=dict,
        description="Domain-specific ontology parameters"
    )
    schema_version: str = Field(
        default="sem-1.0",
        pattern=r"^sem-\d+\.\d+$",
        description="Contract schema version"
    )

    @field_validator('text')
    @classmethod
    def text_not_empty(cls, v: str) -> str:
        """Ensure text is not just whitespace."""
        if not v or not v.strip():
            raise ValueError("text must contain non-whitespace characters")
        return v

class SemanticAnalyzerOutputModel(StrictModel):
    """Runtime validator for SemanticAnalyzerOutputContract."""
    semantic_cube: dict[str, Any] = Field(description="Semantic analysis results")
    coherence_score: float = Field(ge=0.0, le=1.0, description="Coherence metric")
    complexity_score: float = Field(ge=0.0, description="Complexity metric")
    domain_classification: dict[str, float] = Field(
        description="Domain probability distribution"
    )
    schema_version: str = Field(
        default="sem-1.0",
        pattern=r"^sem-\d+\.\d+$"
    )

    @field_validator('domain_classification')
    @classmethod
    def validate_probabilities(cls, v: dict[str, float]) -> dict[str, float]:
        """Ensure all domain probabilities are in [0, 1]."""
        for domain, prob in v.items():
            if not (0.0 <= prob <= 1.0):
                raise ValueError(f"Probability for {domain} must be in [0, 1], got")

```

```
{prob}")
    return v
```

```
# =====
# DERECK_BEACH.PY CONTRACTS
# =====
```

```
class CDAFFrameworkInputModel(StrictModel):
    """Runtime validator for CDAFFrameworkInputContract."""
    document_text: str = Field(min_length=1, description="Document to analyze")
    plan_metadata: dict[str, Any] = Field(
        default_factory=dict,
        description="Metadata about the plan"
    )
    config: dict[str, Any] = Field(
        default_factory=dict,
        description="Framework configuration"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")
```

```
class CDAFFrameworkOutputModel(StrictModel):
    """Runtime validator for CDAFFrameworkOutputContract."""
    causal_mechanisms: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Identified causal mechanisms"
    )
    evidential_tests: dict[str, Any] = Field(
        default_factory=dict,
        description="Statistical test results"
    )
    bayesian_inference: dict[str, Any] = Field(
        default_factory=dict,
        description="Bayesian analysis results"
    )
    audit_results: dict[str, Any] = Field(
        default_factory=dict,
        description="Audit findings and recommendations"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")
```

```
# =====
# FINANCIERO_VIABILIDAD_TABLAS.PY CONTRACTS
# =====
```

```
class PDETAAnalyzerInputModel(StrictModel):
    """Runtime validator for PDETAAnalyzerInputContract."""
    document_content: str = Field(min_length=1, description="Document content")
    extract_tables: bool = Field(default=True, description="Whether to extract tables")
    config: dict[str, Any] = Field(default_factory=dict, description="Configuration")
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")
```

```
class PDETAAnalyzerOutputModel(StrictModel):
    """Runtime validator for PDETAAnalyzerOutputContract."""
    extracted_tables: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Extracted financial tables"
    )
    financial_indicators: dict[str, float] = Field(
        default_factory=dict,
        description="Calculated financial metrics"
    )
    viability_score: float = Field(
        ge=0.0, le=1.0,
        description="Overall viability score"
    )
    quality_scores: dict[str, float] = Field(
        default_factory=dict,
        description="Quality assessment scores"
    )
```

```

)
schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# =====
# TEORIA_CAMBIO.PY CONTRACTS
# =====

class TeoriaCambioInputModel(StrictModel):
    """Runtime validator for TeoriaCambioInputContract."""
    document_text: str = Field(min_length=1, description="Document to analyze")
    strategic_goals: list[str] = Field(
        default_factory=list,
        description="Identified strategic goals"
    )
    config: dict[str, Any] = Field(default_factory=dict, description="Configuration")
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class TeoriaCambioOutputModel(StrictModel):
    """Runtime validator for TeoriaCambioOutputContract."""
    causal_dag: dict[str, Any] = Field(
        default_factory=dict,
        description="Causal directed acyclic graph"
    )
    validation_results: dict[str, Any] = Field(
        default_factory=dict,
        description="Model validation results"
    )
    monte_carlo_results: dict[str, Any] | None = Field(
        default=None,
        description="Monte Carlo simulation results"
    )
    graph_visualizations: list[dict[str, Any]] | None = Field(
        default=None,
        description="Graph visualization data"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# =====
# CONTRADICTION_DETECCION.PY CONTRACTS
# =====

class ContradictionDetectorInputModel(StrictModel):
    """Runtime validator for ContradictionDetectorInputContract."""
    text: str = Field(min_length=1, description="Text to analyze for contradictions")
    plan_name: str = Field(min_length=1, description="Name of the plan")
    dimension: str | None = Field(
        default=None,
        description="PolicyDimension enum value"
    )
    config: dict[str, Any] = Field(default_factory=dict, description="Configuration")
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class ContradictionDetectorOutputModel(StrictModel):
    """Runtime validator for ContradictionDetectorOutputContract."""
    contradictions: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Detected contradictions"
    )
    confidence_scores: dict[str, float] = Field(
        default_factory=dict,
        description="Confidence in each detection"
    )
    temporal_conflicts: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Temporal inconsistencies"
    )
    severity_scores: dict[str, float] = Field(
        default_factory=dict,

```

```

        description="Severity ratings"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# =====
# EMBEDDING_POLICY.PY CONTRACTS
# =====

class EmbeddingPolicyInputModel(StrictModel):
    """Runtime validator for EmbeddingPolicyInputContract."""
    text: str = Field(min_length=1, description="Text to embed")
    dimensions: list[str] = Field(
        default_factory=list,
        description="Policy dimensions to analyze"
    )
    embedding_model_config: dict[str, Any] = Field(
        default_factory=dict,
        description="Embedding model configuration",
        alias="model_config"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class EmbeddingPolicyOutputModel(StrictModel):
    """Runtime validator for EmbeddingPolicyOutputContract."""
    embeddings: list[list[float]] = Field(
        default_factory=list,
        description="Generated embeddings"
    )
    similarity_scores: dict[str, float] = Field(
        default_factory=dict,
        description="Similarity metrics"
    )
    bayesian_evaluation: dict[str, Any] = Field(
        default_factory=dict,
        description="Bayesian evaluation results"
    )
    policy_metrics: dict[str, float] = Field(
        default_factory=dict,
        description="Policy-specific metrics"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# =====
# SEMANTIC_CHUNKING_POLICY.PY CONTRACTS
# =====

class SemanticChunkingInputModel(StrictModel):
    """Runtime validator for SemanticChunkingInputContract."""
    text: str = Field(min_length=1, description="Text to chunk")
    preserve_structure: bool = Field(
        default=True,
        description="Whether to preserve document structure"
    )
    config: dict[str, Any] = Field(default_factory=dict, description="Configuration")
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class SemanticChunkingOutputModel(StrictModel):
    """Runtime validator for SemanticChunkingOutputContract."""
    chunks: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Semantic chunks"
    )
    causal_dimensions: dict[str, dict[str, Any]] = Field(
        default_factory=dict,
        description="Causal dimension analysis"
    )
    key_excerpts: dict[str, list[str]] = Field(
        default_factory=dict,

```

```

        description="Key excerpts by category"
    )
    summary: dict[str, Any] = Field(
        default_factory=dict,
        description="Summary statistics"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# =====
# POLICY_PROCESSOR.PY CONTRACTS
# =====

class PolicyProcessorInputModel(StrictModel):
    """Runtime validator for PolicyProcessorInputContract."""
    data: Any = Field(description="Raw data to process")
    text: str = Field(min_length=1, description="Text content")
    sentences: list[str] = Field(
        default_factory=list,
        description="Pre-segmented sentences"
    )
    tables: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Extracted tables"
    )
    config: dict[str, Any] = Field(default_factory=dict, description="Configuration")
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class PolicyProcessorOutputModel(StrictModel):
    """Runtime validator for PolicyProcessorOutputContract."""
    processed_data: dict[str, Any] = Field(
        default_factory=dict,
        description="Processed results"
    )
    evidence_bundles: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Evidence bundles"
    )
    bayesian_scores: dict[str, float] = Field(
        default_factory=dict,
        description="Bayesian scores"
    )
    matched_patterns: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Matched patterns"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# =====
# EXPORTS
# =====

__all__ = [
    # Base
    'StrictModel',

    # Analyzer_one
    'SemanticAnalyzerInputModel',
    'SemanticAnalyzerOutputModel',

    # derek_beach
    'CDAFFrameworkInputModel',
    'CDAFFrameworkOutputModel',

    # financiero_viabilidad_tablas
    'PDETAAnalyzerInputModel',
    'PDETAAnalyzerOutputModel',

    # teoria_cambio

```

```

'TeoriaCambioInputModel',
'TeoriaCambioOutputModel',

# contradiction_deteccion
'ContradictionDetectorInputModel',
'ContradictionDetectorOutputModel',

# embedding_policy
'EmbeddingPolicyInputModel',
'EmbeddingPolicyOutputModel',

# semantic_chunking_policy
'SemanticChunkingInputModel',
'SemanticChunkingOutputModel',

# policy_processor
'PolicyProcessorInputModel',
'PolicyProcessorOutputModel',
]

```

===== FILE: src/saaaaaa/utils/core\_contracts.py =====

"""

Core Module Contracts - Type-safe API boundaries for pure library modules.

This module defines InputContract and OutputContract TypedDicts for each core module to establish clear API boundaries and enable dependency injection.

Architectural Principles:

- Core modules receive all data via InputContract parameters
- Core modules return data via OutputContract structures
- No I/O operations within core modules
- All I/O happens in orchestrator/factory.py
- Type-safe contracts with strict typing

Version: 1.1.0

Schema Version: sem-1.0 (initial stable release)

Status: Active - Runtime validation available in contracts\_runtime.py

"""

```

from typing import Any, TypedDict
from saaaaaa.core.calibration.decorators import calibrated_method

```

try:

```

    from typing import NotRequired # Python 3.11+
except ImportError:
    from typing_extensions import NotRequired # Python 3.9-3.10

```

```

# =====
# ANALYZER_ONE.PY CONTRACTS
# =====

```

```

class SemanticAnalyzerInputContract(TypedDict):
    """Input contract for SemanticAnalyzer methods.

```

Example:

```

    {
        "text": "El plan de desarrollo municipal...",
        "segments": ["Segment 1", "Segment 2"],
        "ontology_params": {"domain": "municipal"}
    }
    """

```

```

text: str
segments: NotRequired[list[str]]
ontology_params: NotRequired[dict[str, Any]]

```

```

class SemanticAnalyzerOutputContract(TypedDict):
    """Output contract for SemanticAnalyzer methods."""
    semantic_cube: dict[str, Any]

```

```

coherence_score: float
complexity_score: float
domain_classification: dict[str, float]

# =====
# DERECK_BEACH.PY CONTRACTS
# =====

class CDAFFrameworkInputContract(TypedDict):
    """Input contract for CDAFFramework (Causal Deconstruction Audit Framework)."""
    document_text: str
    plan_metadata: dict[str, Any]
    config: NotRequired[dict[str, Any]]

class CDAFFrameworkOutputContract(TypedDict):
    """Output contract for CDAFFramework."""
    causal_mechanisms: list[dict[str, Any]]
    evidential_tests: dict[str, Any]
    bayesian_inference: dict[str, Any]
    audit_results: dict[str, Any]

# =====
# FINANCIERO_VIABILIDAD_TABLAS.PY CONTRACTS
# =====

class PDETAnalyzerInputContract(TypedDict):
    """Input contract for PDET (Programas de Desarrollo con Enfoque Territorial) Analyzer."""
    document_content: str
    extract_tables: NotRequired[bool]
    config: NotRequired[dict[str, Any]]

class PDETAnalyzerOutputContract(TypedDict):
    """Output contract for PDET Analyzer."""
    extracted_tables: list[dict[str, Any]]
    financial_indicators: dict[str, float]
    viability_score: float
    quality_scores: dict[str, float]

# =====
# TEORIA_CAMBIO.PY CONTRACTS
# =====

class TeoriaCambioInputContract(TypedDict):
    """Input contract for Theory of Change analysis."""
    document_text: str
    strategic_goals: NotRequired[list[str]]
    config: NotRequired[dict[str, Any]]

class TeoriaCambioOutputContract(TypedDict):
    """Output contract for Theory of Change analysis."""
    causal_dag: dict[str, Any]
    validation_results: dict[str, Any]
    monte_carlo_results: NotRequired[dict[str, Any]]
    graph_visualizations: NotRequired[list[dict[str, Any]]]

# =====
# CONTRADICTION_DETECCION.PY CONTRACTS
# =====

class ContradictionDetectorInputContract(TypedDict):
    """Input contract for PolicyContradictionDetector."""
    text: str
    plan_name: str
    dimension: NotRequired[str] # PolicyDimension enum value
    config: NotRequired[dict[str, Any]]

class ContradictionDetectorOutputContract(TypedDict):

```

```

"""Output contract for PolicyContradictionDetector."""
contradictions: list[dict[str, Any]]
confidence_scores: dict[str, float]
temporal_conflicts: list[dict[str, Any]]
severity_scores: dict[str, float]

# =====
# EMBEDDING_POLICY.PY CONTRACTS
# =====

class EmbeddingPolicyInputContract(TypedDict):
    """Input contract for embedding-based policy analysis."""
    text: str
    dimensions: NotRequired[list[str]]
    model_config: NotRequired[dict[str, Any]]

class EmbeddingPolicyOutputContract(TypedDict):
    """Output contract for embedding policy analysis."""
    embeddings: list[list[float]]
    similarity_scores: dict[str, float]
    bayesian_evaluation: dict[str, Any]
    policy_metrics: dict[str, float]

# =====
# SEMANTIC_CHUNKING_POLICY.PY CONTRACTS
# =====

class SemanticChunkingInputContract(TypedDict):
    """Input contract for semantic chunking and policy document analysis."""
    text: str
    preserve_structure: NotRequired[bool]
    config: NotRequired[dict[str, Any]]

class SemanticChunkingOutputContract(TypedDict):
    """Output contract for semantic chunking."""
    chunks: list[dict[str, Any]]
    causal_dimensions: dict[str, dict[str, Any]]
    key_excerpts: dict[str, list[str]]
    summary: dict[str, Any]

# =====
# POLICY_PROCESSOR.PY CONTRACTS
# =====

class PolicyProcessorInputContract(TypedDict):
    """Input contract for IndustrialPolicyProcessor."""
    data: Any
    text: str
    sentences: NotRequired[list[str]]
    tables: NotRequired[list[dict[str, Any]]]
    config: NotRequired[dict[str, Any]]

class PolicyProcessorOutputContract(TypedDict):
    """Output contract for IndustrialPolicyProcessor."""
    processed_data: dict[str, Any]
    evidence_bundles: list[dict[str, Any]]
    bayesian_scores: dict[str, float]
    matched_patterns: list[dict[str, Any]]

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

class DocumentData(TypedDict):
    """Standard document data structure from saaaaaa.core.orchestrator.

    This is what the orchestrator/factory provides to core modules.
    """

```

```

raw_text: str
sentences: list[str]
tables: list[dict[str, Any]]
metadata: dict[str, Any]

__all__ = [
    # Analyzer_one
    'SemanticAnalyzerInputContract',
    'SemanticAnalyzerOutputContract',

    # derek_beach
    'CDAFFrameworkInputContract',
    'CDAFFrameworkOutputContract',

    # financiero_viabilidad_tablas
    'PDEAnalyzerInputContract',
    'PDEAnalyzerOutputContract',

    # teoria_cambio
    'TeoriaCambioInputContract',
    'TeoriaCambioOutputContract',

    # contradiction_deteccion
    'ContradictionDetectorInputContract',
    'ContradictionDetectorOutputContract',

    # embedding_policy
    'EmbeddingPolicyInputContract',
    'EmbeddingPolicyOutputContract',

    # semantic_chunking_policy
    'SemanticChunkingInputContract',
    'SemanticChunkingOutputContract',

    # policy_processor
    'PolicyProcessorInputContract',
    'PolicyProcessorOutputContract',

    # Shared
    'DocumentData',
]

```

===== FILE: src/saaaaaa/utils/coverage\_gate.py =====

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

Coverage Enforcement Gate

=====

Enforces hard-fail at <555 methods threshold + audit.json emission

Requirements:

- Count all public methods across Producer classes
- Generate audit.json with method counts and validation results
- Hard-fail if total methods < 555
- Include schema validation results

"""

```

import ast
import json
import sys
from datetime import datetime
from pathlib import Path
from saaaaaa.core.calibration.decorators import calibrated_method

```

```

def count_methods_in_class(filepath: Path, class_name: str) -> tuple[list[str], dict[str,
int]]:

```

```

    """Count public and private methods in a class and return method names"""

```

```

    if not filepath.exists():

```

```

    return [], {"public": 0, "private": 0, "total": 0}

with open(filepath, encoding='utf-8') as f:
    tree = ast.parse(f.read())

method_names = []
method_counts = {
    "public": 0,
    "private": 0,
    "total": 0
}

for node in ast.walk(tree):
    if isinstance(node, ast.ClassDef) and node.name == class_name:
        for item in node.body:
            if isinstance(item, ast.FunctionDef):
                method_names.append(item.name)
                if not item.name.startswith('_'):
                    method_counts["public"] += 1
                else:
                    method_counts["private"] += 1
                method_counts["total"] += 1

return method_names, method_counts

def validate_schema_exists(module_dir: Path) -> tuple[bool, list[str]]:
    """Validate that schema files exist for a module"""
    if not module_dir.exists():
        return False, []

    schema_files = list(module_dir.glob("*.schema.json"))
    return len(schema_files) > 0, [f.name for f in schema_files]

def count_file_methods(filepath: Path) -> tuple[int, int]:
    """Count all public and total methods in a file"""
    if not filepath.exists():
        return 0, 0

    with open(filepath, encoding='utf-8') as f:
        try:
            tree = ast.parse(f.read())
            public_methods = 0
            all_methods = 0

            for node in ast.walk(tree):
                if isinstance(node, ast.FunctionDef):
                    all_methods += 1
                    if not node.name.startswith('_'):
                        public_methods += 1

            return public_methods, all_methods
        except Exception as e:
            print(f"Error parsing {filepath}: {e}")
            return 0, 0

def count_all_methods() -> dict[str, any]:
    """Count all methods across all modules and producers"""

    # All files to analyze
    files_to_analyze = [
        "financiero_viabilidad_tablas.py",
        "Analyzer_one.py",
        "contradiction_deteccion.py",
        "embedding_policy.py",
        "teoria_cambio.py",
        "derek_beach.py",
        "policy_processor.py",
        "report_assembly.py",
    ]

```

```

    "semantic_chunking_policy.py"
]

# Producer classes to check
producers = {
    "SemanticChunkingProducer": "semantic_chunking_policy.py",
    "EmbeddingPolicyProducer": "embedding_policy.py",
    "DerekBeachProducer": "derek_beach.py",
    "ReportAssemblyProducer": "report_assembly.py"
}

results = {
    "timestamp": datetime.now().isoformat(),
    "files": {},
    "producers": {},
    "totals": {
        "file_public_methods": 0,
        "file_total_methods": 0,
        "producer_methods": 0,
        "threshold": 555,
        "meets_threshold": False
    },
    "schema_validation": {},
    "audit_status": "PENDING"
}

# Count file methods
print("=" * 80)
print("FILE METHOD COUNTS")
print("=" * 80)

for filepath_str in files_to_analyze:
    filepath = Path(filepath_str)
    public_methods, total_methods = count_file_methods(filepath)
    results["files"][filepath_str] = {
        "public_methods": public_methods,
        "total_methods": total_methods
    }
    results["totals"]["file_public_methods"] += public_methods
    results["totals"]["file_total_methods"] += total_methods
    print(f'{filepath_str:45} | {public_methods:4} public | {total_methods:4} total')

# Count Producer methods
print("\n" + "=" * 80)
print("PRODUCER METHOD COUNTS")
print("=" * 80)

for class_name, filepath in producers.items():
    methods, counts = count_methods_in_class(Path(filepath), class_name)
    results["producers"][class_name] = {
        "file": filepath,
        "methods": methods,
        "counts": counts,
        "public_methods": counts["public"]
    }
    results["totals"]["producer_methods"] += counts["public"]
    print(f'{class_name:45} | {counts["public"]:3} public | {counts["private"]:3} private | {counts["total"]:3} total')

# Update meets_threshold
results["totals"]["meets_threshold"] = (
    results["totals"]["file_total_methods"] >= 555
)

# Validate schemas
print("\n" + "=" * 80)
print("SCHEMA VALIDATION")
print("=" * 80)

```

```

schema_modules = [
    "semantic_chunking_policy",
    "embedding_policy",
    "derek_beach",
    "report_assembly"
]

for module in schema_modules:
    module_dir = Path("schemas") / module
    has_schemas, schema_files = validate_schema_exists(module_dir)
    results["schema_validation"][module] = {
        "has_schemas": has_schemas,
        "schema_files": schema_files,
        "schema_count": len(schema_files)
    }
    status = "✓" if has_schemas else "✗"
    print(f"{module:35} | {status} | {len(schema_files)} schemas")

# Determine audit status
all_have_schemas = all(
    v["has_schemas"] for v in results["schema_validation"].values()
)

if results["totals"]["meets_threshold"] and all_have_schemas:
    results["audit_status"] = "PASS"
else:
    results["audit_status"] = "FAIL"

return results

def main() -> int:
    """Main entry point"""
    print("\n" + "=" * 80)
    print("COVERAGE ENFORCEMENT GATE")
    print("=" * 80 + "\n")

    # Count all methods
    results = count_all_methods()

    # Print summary
    print("\n" + "=" * 80)
    print("SUMMARY")
    print("=" * 80)
    print(f"Total file methods:      {results['totals']['file_total_methods']:4}")
    print(f"Total public methods:      {results['totals']['file_public_methods']:4}")
    print(f"Producer methods:          {results['totals']['producer_methods']:4}")
    print(f"Threshold:                  {results['totals']['threshold']:4}")
    print(f"Meets threshold:            {results['totals']['meets_threshold']}")
    print(f"All schemas present:       {all(v['has_schemas'] for v in
results['schema_validation'].values())}")
    print(f"Audit status:              {results['audit_status']}")

    # Save audit.json
    audit_path = Path("audit.json")
    with open(audit_path, 'w', encoding='utf-8') as f:
        json.dump(results, f, indent=2)

    print(f"\n✓ Audit results saved to {audit_path}")

    # Enforce hard-fail
    if not results["totals"]["meets_threshold"]:
        print("\n" + "=" * 80)
        print("✗ COVERAGE GATE FAILED")
        print("=" * 80)
        print(f"Required: {results['totals']['threshold']} methods")
        print(f"Found:    {results['totals']['file_total_methods']} methods")
        print(f"Gap:      {results['totals']['threshold'] -")

```

```

results['totals']['file_total_methods']} methods")
    print("=" * 80 + "\n")
    return 1

# Check schema validation
if not all(v['has_schemas'] for v in results['schema_validation'].values()):
    print("\n" + "=" * 80)
    print("✘ SCHEMA VALIDATION FAILED")
    print("=" * 80)
    for module, validation in results['schema_validation'].items():
        if not validation['has_schemas']:
            print(f"Missing schemas for: {module}")
    print("=" * 80 + "\n")
    return 1

print("\n" + "=" * 80)
print("✓ COVERAGE GATE PASSED")
print("=" * 80)
print(f"All {results['totals']['file_total_methods']} methods accounted for")
print(f"{results['totals']['file_public_methods']} public methods available")
print(f"{results['totals']['producer_methods']} producer methods exposed")
print("All schema contracts validated")
print("=" * 80 + "\n")

return 0

if __name__ == "__main__":
    sys.exit(main())

===== FILE: src/saaaaaa/utils/cpp_adapter.py =====
"""CPP to Orchestrator Adapter.

This adapter converts Canon Policy Package (CPP) documents from the ingestion pipeline
into the orchestrator's PreprocessedDocument format.

Note: This is the canonical adapter implementation. SPC (Smart Policy Chunks) is the
precursor to CPP.

Design Principles:
- Preserves complete provenance information
- Orders chunks by text_span.start for deterministic ordering
- Computes provenance_completeness metric
- Provides prescriptive error messages on failure
- Supports micro, meso, and macro chunk resolutions
- Optional dependencies handled gracefully (pyarrow, structlog)
"""

from __future__ import annotations

import logging
from datetime import datetime, timezone
from types import MappingProxyType
from typing import Any

from saaaaaa.core.orchestrator.core import PreprocessedDocument
from saaaaaa import get_parameter_loader
from saaaaaa.core.calibration.decorators import calibrated_method

logger = logging.getLogger(__name__)

_EMPTY_MAPPING = MappingProxyType({})

class CPPAdapterError(Exception):
    """Raised when CPP to PreprocessedDocument conversion fails."""
    pass

```

class CPPAdapter:

"""

Adapter to convert CanonPolicyPackage (CPP output) to PreprocessedDocument.

This is the canonical adapter for the FARFAN pipeline, converting the rich CanonPolicyPackage data into the format expected by the orchestrator.

"""

def \_\_init\_\_(self, enable\_runtime\_validation: bool = True) -> None:

"""Initialize the CPP adapter.

Args:

enable\_runtime\_validation: Enable WiringValidator for runtime contract checking

"""

self.logger = logging.getLogger(self.\_\_class\_\_.\_\_name\_\_)

# Initialize WiringValidator for runtime contract validation

self.enable\_runtime\_validation = enable\_runtime\_validation

if enable\_runtime\_validation:

try:

from saaaaaa.core.wiring.validation import WiringValidator

self.wiring\_validator = WiringValidator()

self.logger.info("WiringValidator enabled for runtime contract checking")

except ImportError:

self.logger.warning(

"WiringValidator not available. Runtime validation disabled."

)

self.wiring\_validator = None

else:

self.wiring\_validator = None

def to\_preprocessed\_document(

self,

canon\_package: Any,

document\_id: str

) -> PreprocessedDocument:

"""

Convert CanonPolicyPackage to PreprocessedDocument.

Args:

canon\_package: CanonPolicyPackage from ingestion

document\_id: Unique document identifier

Returns:

PreprocessedDocument ready for orchestrator

Raises:

CPPAdapterError: If conversion fails or data is invalid

CanonPolicyPackage Expected Attributes:

Required:

- chunk\_graph: ChunkGraph with .chunks dict

- chunk\_graph.chunks: dict of chunk objects with .text and .text\_span

Optional (handled with hasattr checks):

- schema\_version: str (default: 'SPC-2025.1')

- quality\_metrics: object with metrics like provenance\_completeness,

- structural\_consistency, boundary\_f1, kpi\_linkage\_rate,

- budget\_consistency\_score, temporal\_robustness, chunk\_context\_coverage

- policy\_manifest: object with axes, programs, projects, years,

territories

- metadata: dict with optional 'spc\_rich\_data' key

Chunk Optional Attributes (handled with hasattr checks):

- entities: list of entity objects with .text attribute

- time\_facets: object with .years list

- budget: object with amount, currency, year, use, source attributes

```

"""
self.logger.info(f"Converting CanonPolicyPackage to PreprocessedDocument:
{document_id}")

# === COMPREHENSIVE VALIDATION PHASE (H1.5) ===
# 6-layer validation for robust phase-one output processing

# V1: Validate canon_package exists
if not canon_package:
    raise CPPAdapterError(
        "canon_package is None or empty. "
        "Ensure ingestion completed successfully."
    )

# V2: Validate document_id
if not document_id or not isinstance(document_id, str) or not document_id.strip():
    raise CPPAdapterError(
        f"document_id must be a non-empty string. "
        f"Received: {repr(document_id)}"
    )

# V3: Validate chunk_graph exists
if not hasattr(canon_package, 'chunk_graph') or not canon_package.chunk_graph:
    raise CPPAdapterError(
        "canon_package must have a valid chunk_graph. "
        "Check that SmartChunkConverter produced valid output."
    )

chunk_graph = canon_package.chunk_graph

# V4: Validate chunks dict is non-empty
if not chunk_graph.chunks:
    raise CPPAdapterError(
        "chunk_graph.chunks is empty - no chunks to process. "
        "Minimum 1 chunk required from phase-one."
    )

# V5: Validate individual chunks have required attributes
validation_failures = []
for chunk_id, chunk in chunk_graph.chunks.items():
    if not hasattr(chunk, 'text'):
        validation_failures.append(f"Chunk {chunk_id}: missing 'text' attribute")
    elif not chunk.text or not chunk.text.strip():
        validation_failures.append(f"Chunk {chunk_id}: text is empty or
whitespace")

    if not hasattr(chunk, 'text_span'):
        validation_failures.append(f"Chunk {chunk_id}: missing 'text_span'
attribute")
    elif not hasattr(chunk.text_span, 'start') or not hasattr(chunk.text_span,
'end'):
        validation_failures.append(f"Chunk {chunk_id}: invalid text_span (missing
start/end)")

# V6: Report validation failures with context
if validation_failures:
    failure_summary = "\n - ".join(validation_failures)
    raise CPPAdapterError(
        f"Chunk validation failed ({len(validation_failures)} errors):\n -
{failure_summary}\n"
        f"Total chunks: {len(chunk_graph.chunks)}\n"
        f"This indicates SmartChunkConverter produced invalid output."
    )

# Sort chunks by document position for deterministic ordering
sorted_chunks = sorted(
    chunk_graph.chunks.values(),
    key=lambda c: c.text_span.start if hasattr(c, 'text_span') and c.text_span

```

```

else 0
)

self.logger.info(f"Processing {len(sorted_chunks)} chunks")

# Build full text by concatenating chunks
full_text_parts: list[str] = []
sentences: list[dict[str, Any]] = []
sentence_metadata: list[dict[str, Any]] = []
tables: list[dict[str, Any]] = []
chunk_index: dict[str, int] = {}
chunk_summaries: list[dict[str, Any]] = []

# Track indices for building indexes
term_index: dict[str, list[int]] = {}
numeric_index: dict[str, list[int]] = {}
temporal_index: dict[str, list[int]] = {}
entity_index: dict[str, list[int]] = {}

# Track running offset that matches how full_text is built
current_offset = 0

provenance_with_data = 0

for idx, chunk in enumerate(sorted_chunks):
    chunk_text = chunk.text
    chunk_start = current_offset
    chunk_index[chunk.id] = idx

    # Add to full text
    full_text_parts.append(chunk_text)

    # Create sentence entry (each chunk is represented as a sentence for
orchestrator compatibility)
    sentences.append(
        {
            "text": chunk_text,
            "chunk_id": chunk.id,
            "resolution": (chunk.resolution.value.lower() if hasattr(chunk,
"resolution") else None),
        }
    )

    # Create chunk metadata for per-sentence tracking
    chunk_end = chunk_start + len(chunk_text)

    # CRITICAL: Preserve PA×DIM metadata for Phase 2 question routing
    extra_metadata = {
        "chunk_id": chunk.id,
        "policy_area_id": chunk.policy_area_id if hasattr(chunk, "policy_area_id")
else None,
        "dimension_id": chunk.dimension_id if hasattr(chunk, "dimension_id") else
None,
        "resolution": chunk.resolution.value.lower() if hasattr(chunk,
"resolution") else None,
    }

    # Add facets if available
    if hasattr(chunk, "policy_facets") and chunk.policy_facets:
        extra_metadata["policy_facets"] = {
            "axes": chunk.policy_facets.axes if hasattr(chunk.policy_facets,
"axes") else [],
            "programs": chunk.policy_facets.programs if
hasattr(chunk.policy_facets, "programs") else [],
            "projects": chunk.policy_facets.projects if
hasattr(chunk.policy_facets, "projects") else [],
        }

```

```

        if hasattr(chunk, "time_facets") and chunk.time_facets:
            extra_metadata["time_facets"] = {
                "years": chunk.time_facets.years if hasattr(chunk.time_facets,
"years") else [],
                "periods": chunk.time_facets.periods if hasattr(chunk.time_facets,
"periods") else [],
            }

        if hasattr(chunk, "geo_facets") and chunk.geo_facets:
            extra_metadata["geo_facets"] = {
                "territories": chunk.geo_facets.territories if
hasattr(chunk.geo_facets, "territories") else [],
                "regions": chunk.geo_facets.regions if hasattr(chunk.geo_facets,
"regions") else [],
            }

        sentence_metadata.append(
            {
                "index": idx,
                "page_number": None,
                "start_char": chunk_start,
                "end_char": chunk_end,
                "extra": dict(extra_metadata),
            }
        )

        chunk_summary = {
            "id": chunk.id,
            "resolution": (chunk.resolution.value.lower() if hasattr(chunk,
"resolution") else None),
            "text_span": {"start": chunk_start, "end": chunk_end},
            "policy_area_id": extra_metadata["policy_area_id"],
            "dimension_id": extra_metadata["dimension_id"],
            "has_kpi": hasattr(chunk, "kpi") and chunk.kpi is not None,
            "has_budget": hasattr(chunk, "budget") and chunk.budget is not None,
            "confidence": {
                "layout": getattr(chunk.confidence, "layout", get_parameter_loader().g
et("saaaaaa.utils.cpp_adapter.CPPAdapter.__init__").get("auto_param_L256_66", 0.0)) if
hasattr(chunk, "confidence") else get_parameter_loader().get("saaaaaa.utils.cpp_adapter.CP
PAdapter.__init__").get("auto_param_L256_108", 0.0),
                "ocr": getattr(chunk.confidence, "ocr", get_parameter_loader().get("sa
aaaaa.utils.cpp_adapter.CPPAdapter.__init__").get("auto_param_L257_60", 0.0)) if
hasattr(chunk, "confidence") else get_parameter_loader().get("saaaaaa.utils.cpp_adapter.CP
PAdapter.__init__").get("auto_param_L257_102", 0.0),
                "typing": getattr(chunk.confidence, "typing", get_parameter_loader().g
et("saaaaaa.utils.cpp_adapter.CPPAdapter.__init__").get("auto_param_L258_66", 0.0)) if
hasattr(chunk, "confidence") else get_parameter_loader().get("saaaaaa.utils.cpp_adapter.CP
PAdapter.__init__").get("auto_param_L258_108", 0.0),
            },
        }
        chunk_summaries.append(chunk_summary)

        if hasattr(chunk, "provenance") and chunk.provenance:
            provenance_with_data += 1

        # Advance offset by chunk length + 1 space separator
        current_offset = chunk_end + 1

        # Extract entities for entity_index
        if hasattr(chunk, "entities") and chunk.entities:
            for entity in chunk.entities:
                entity_text = entity.text if hasattr(entity, "text") else str(entity)
                if entity_text not in entity_index:
                    entity_index[entity_text] = []
                entity_index[entity_text].append(idx)

        # Extract temporal markers for temporal_index
        if hasattr(chunk, "time_facets") and chunk.time_facets:

```

```

        if hasattr(chunk.time_facets, "years") and chunk.time_facets.years:
            for year in chunk.time_facets.years:
                year_key = str(year)
                if year_key not in temporal_index:
                    temporal_index[year_key] = []
                temporal_index[year_key].append(idx)

    # Extract budget for tables
    if hasattr(chunk, "budget") and chunk.budget:
        budget = chunk.budget
        tables.append(
            {
                "table_id": f"budget_{idx}",
                "label": f"Budget: {budget.source if hasattr(budget, 'source')
else 'Unknown'}",
                "amount": getattr(budget, "amount", 0),
                "currency": getattr(budget, "currency", "COP"),
                "year": getattr(budget, "year", None),
                "use": getattr(budget, "use", None),
                "source": getattr(budget, "source", None),
            }
        )

    # Join full text
    full_text = " ".join(full_text_parts)

    if not full_text:
        raise CPPAdapterError("Generated full_text is empty")

    # Build document indexes
    indexes = {
        "term_index": {k: tuple(v) for k, v in term_index.items()},
        "numeric_index": {k: tuple(v) for k, v in numeric_index.items()},
        "temporal_index": {k: tuple(v) for k, v in temporal_index.items()},
        "entity_index": {k: tuple(v) for k, v in entity_index.items()},
    }

    # Build metadata from canon_package
    metadata_dict = {
        'adapter_source': 'CPPAdapter',
        'schema_version': canon_package.schema_version if hasattr(canon_package,
'schema_version') else 'SPC-2025.1',
        'chunk_count': len(sorted_chunks),
        'processing_mode': 'chunked',
        'chunks': chunk_summaries,
    }

    # Add quality metrics if available
    if hasattr(canon_package, 'quality_metrics') and canon_package.quality_metrics:
        qm = canon_package.quality_metrics
        metadata_dict['quality_metrics'] = {
            'provenance_completeness': qm.provenance_completeness if hasattr(qm,
'provenance_completeness') else get_parameter_loader().get("saaaaaa.utils.cpp_adapter.CPPA
dapter.__init__").get("auto_param_L328_117", 0.0),
            'structural_consistency': qm.structural_consistency if hasattr(qm,
'structural_consistency') else get_parameter_loader().get("saaaaaa.utils.cpp_adapter.CPPAd
apter.__init__").get("auto_param_L329_114", 0.0),
            'boundary_f1': qm.boundary_f1 if hasattr(qm, 'boundary_f1') else get_param
eter_loader().get("saaaaaa.utils.cpp_adapter.CPPAdapter.__init__").get("auto_param_L330_81
", 0.0),
            'kpi_linkage_rate': qm.kpi_linkage_rate if hasattr(qm, 'kpi_linkage_rate')
else get_parameter_loader().get("saaaaaa.utils.cpp_adapter.CPPAdapter.__init__").get("aut
o_param_L331_96", 0.0),
            'budget_consistency_score': qm.budget_consistency_score if hasattr(qm,
'budget_consistency_score') else get_parameter_loader().get("saaaaaa.utils.cpp_adapter.CPP
Adapter.__init__").get("auto_param_L332_120", 0.0),
            'temporal_robustness': qm.temporal_robustness if hasattr(qm,
'temporal_robustness') else get_parameter_loader().get("saaaaaa.utils.cpp_adapter.CPPAdapt

```

```

er.__init__(self).get("auto_param_L333_105", 0.0),
    'chunk_context_coverage': qm.chunk_context_coverage if hasattr(qm,
'chunk_context_coverage') else get_parameter_loader().get("saaaaaa.utils.cpp_adapter.CPPAd
apter.__init__(self).get("auto_param_L334_114", 0.0),
    }

# Add policy manifest if available
if hasattr(canon_package, 'policy_manifest') and canon_package.policy_manifest:
    pm = canon_package.policy_manifest
    metadata_dict['policy_manifest'] = {
        'axes': pm.axes if hasattr(pm, 'axes') else [],
        'programs': pm.programs if hasattr(pm, 'programs') else [],
        'projects': pm.projects if hasattr(pm, 'projects') else [],
        'years': pm.years if hasattr(pm, 'years') else [],
        'territories': pm.territories if hasattr(pm, 'territories') else [],
    }

# Add SPC rich data if available in metadata
if hasattr(canon_package, 'metadata') and canon_package.metadata:
    if 'spc_rich_data' in canon_package.metadata:
        metadata_dict['spc_rich_data'] = canon_package.metadata['spc_rich_data']

if len(sorted_chunks) > 0:
    metadata_dict['provenance_completeness'] = provenance_with_data /
len(sorted_chunks)

metadata = MappingProxyType(metadata_dict)

# Detect language (default to Spanish for Colombian policy documents)
language = "es"

# Create PreprocessedDocument (canonical orchestrator dataclass)
preprocessed_doc = PreprocessedDocument(
    document_id=document_id,
    raw_text=full_text,
    sentences=sentences,
    tables=tables,
    metadata=dict(metadata),
    sentence_metadata=sentence_metadata,
    indexes=indexes,
    structured_text={"full_text": full_text, "sections": (), "page_boundaries":
()),
    language=language,
    ingested_at=datetime.now(timezone.utc),
    full_text=full_text,
    chunks=[],
    chunk_index=chunk_index,
    chunk_graph={
        "chunks": {cid: chunk_index[cid] for cid in chunk_index},
        "edges": list(getattr(chunk_graph, "edges", [])),
    },
    processing_mode="chunked",
)

self.logger.info(
    f"Conversion complete: {len(sentences)} sentences, "
    f"{len(tables)} tables, {len(entity_index)} entities indexed"
)

# RUNTIME VALIDATION: Validate Adapter → Orchestrator contract
if self.wiring_validator is not None:
    self.logger.info("Validating Adapter → Orchestrator contract (runtime)")
    try:
        # Convert PreprocessedDocument to dict for validation
        preprocessed_dict = {
            "document_id": preprocessed_doc.document_id,
            "sentence_metadata": preprocessed_doc.sentence_metadata,
            "resolution_index": {}, # Placeholder, as it's not generated by the

```

```

adapter
    "provenance_completeness":
metadata_dict.get('provenance_completeness', get_parameter_loader().get("saaaaaa.utils.cpp
_adapter.CPPAdapter.__init__").get("auto_param_L397_92", 0.0)),
    }
    self.wiring_validator.validate_adapter_to_orchestrator(preprocessed_dict)
    self.logger.info("✓ Adapter → Orchestrator contract validation passed")
except Exception as e:
    self.logger.error(f"Adapter → Orchestrator contract validation failed:
{e}")
    raise ValueError(
        f"Runtime contract violation at Adapter → Orchestrator boundary: {e}"
    ) from e

return preprocessed_doc

```

```

def adapt_cpp_to_orchestrator(
    canon_package: Any,
    document_id: str
) -> PreprocessedDocument:
    """
    Convenience function to adapt CPP to PreprocessedDocument.

    Args:
        canon_package: CanonPolicyPackage from ingestion
        document_id: Unique document identifier

    Returns:
        PreprocessedDocument for orchestrator

    Raises:
        CPPAdapterError: If conversion fails
    """
    adapter = CPPAdapter()
    return adapter.to_preprocessed_document(canon_package, document_id)

```

```

__all__ = [
    'CPPAdapter',
    'CPPAdapterError',
    'adapt_cpp_to_orchestrator',
]

```

```

===== FILE: src/saaaaaa/utils/determinism/__init__.py =====
"""Determinism utilities for reproducible runs."""

```

```

from .seeds import DeterministicContext, SeedFactory

```

```

__all__ = [
    "DeterministicContext",
    "SeedFactory",
]

```

```

===== FILE: src/saaaaaa/utils/determinism/seeds.py =====
"""Deterministic seed management for reproducible execution."""

```

```

from __future__ import annotations

```

```

import hashlib
import os
import random
from dataclasses import dataclass
from typing import TYPE_CHECKING
from saaaaaa.core.calibration.decorators import calibrated_method

```

```

if TYPE_CHECKING:
    from collections.abc import Iterable

```

```

try:
    import numpy as np
    NUMPY_AVAILABLE = True
except ImportError: # pragma: no cover - optional dependency
    np = None # type: ignore
    NUMPY_AVAILABLE = False

class SeedFactory:
    """Factory that derives stable seeds from canonical metadata."""

    DEFAULT_SALT = b"PDM_DETERMINISM_SALT_2025"

    def __init__(self, salt: bytes | None = None) -> None:
        self._salt = salt or self.DEFAULT_SALT

    @calibrated_method("saaaaaa.utils.determinism.seeds.SeedFactory.derive_seed")
    def derive_seed(self, components: Iterable[str]) -> int:
        """Derive a deterministic 32-bit seed from ordered components."""

        material = "".join(str(component) for component in components)
        digest = hashlib.sha256(self._salt + material.encode("utf-8")).digest()
        return int.from_bytes(digest[:4], byteorder="big")

    @calibrated_method("saaaaaa.utils.determinism.seeds.SeedFactory.derive_run_seed")
    def derive_run_seed(self, questionnaire_hash: str, run_id: str) -> int:
        """Derive run-wide seed based on questionnaire hash and run identifier."""

        return self.derive_seed([questionnaire_hash, run_id])

    @calibrated_method("saaaaaa.utils.determinism.seeds.SeedFactory.configure_environment")
    def configure_environment(self, seed: int) -> None:
        """Configure deterministic state for Python and NumPy."""

        os.environ["PYTHONHASHSEED"] = str(seed)
        random.seed(seed)
        if NUMPY_AVAILABLE and np is not None:
            np.random.seed(seed)

    @dataclass
    class DeterministicContext:
        """Deterministic execution context shared with all producers."""

        questionnaire_hash: str
        run_id: str
        seed: int
        numpy_rng: np.random.Generator | None = None

    @calibrated_method("saaaaaa.utils.determinism.seeds.DeterministicContext.apply")
    def apply(self) -> None:
        """Apply deterministic seeding across the runtime environment."""

        os.environ["PYTHONHASHSEED"] = str(self.seed)
        random.seed(self.seed)
        if NUMPY_AVAILABLE and np is not None:
            self.numpy_rng = np.random.default_rng(self.seed)

    @classmethod
    def from_factory(
        cls,
        factory: SeedFactory,
        questionnaire_hash: str,
        run_id: str
    ) -> DeterministicContext:
        seed = factory.derive_run_seed(questionnaire_hash, run_id)
        context = cls(questionnaire_hash=questionnaire_hash, run_id=run_id, seed=seed)
        context.apply()

```

return context

===== FILE: src/saaaaaa/utls/determinism\_helpers.py =====

"""

Determinism Helpers - Centralized Seeding and State Management

=====

Provides centralized determinism enforcement for the entire pipeline:

- Stable seed derivation from policy\_unit\_id and correlation\_id
- Context manager for scoped deterministic execution
- Controls random, numpy.random, and other stochastic libraries

Author: Policy Analytics Research Unit

Version: 1.0.0

License: Proprietary

"""

from \_\_future\_\_ import annotations

import json

import os

import random

from contextlib import contextmanager

from dataclasses import dataclass

from hashlib import sha256

from typing import TYPE\_CHECKING, Any

import numpy as np

from saaaaaa.core.calibration.decorators import calibrated\_method

if TYPE\_CHECKING:

from collections.abc import Iterator

def \_seed\_from(\*parts: Any) -> int:

"""

Derive a 32-bit seed from arbitrary parts via SHA-256.

Args:

\*parts: Components to hash (will be JSON-serialized)

Returns:

32-bit integer seed suitable for random/numpy

Examples:

>>> s1 = \_seed\_from("PU\_123", "corr-1")

>>> s2 = \_seed\_from("PU\_123", "corr-1")

>>> s1 == s2

True

>>> s3 = \_seed\_from("PU\_123", "corr-2")

>>> s1 != s3

True

"""

raw = json.dumps(parts, sort\_keys=True, separators=(",", ":"), ensure\_ascii=False)

# 32-bit seed for numpy/py random

return int(sha256(raw.encode("utf-8")).hexdigest()[:8], 16)

@dataclass(frozen=True)

class Seeds:

"""Container for seeds used in deterministic execution."""

py: int

np: int

@contextmanager

def deterministic(

policy\_unit\_id: str | None = None,

```
correlation_id: str | None = None
) -> Iterator[Seeds]:
"""
Context manager for deterministic execution.

Sets seeds for Python's random and NumPy's random based on
policy_unit_id and correlation_id. Seeds are derived deterministically
via SHA-256 hashing.
```

Args:

```
policy_unit_id: Policy unit identifier (default: env var or "default")
correlation_id: Correlation identifier (default: env var or "run")
```

Yields:

```
Seeds object with py and np seed values
```

Examples:

```
>>> with deterministic("PU_123", "corr-1") as seeds:
...     v1 = random.random()
...     a1 = np.random.rand(3)
>>> with deterministic("PU_123", "corr-1") as seeds:
...     v2 = random.random()
...     a2 = np.random.rand(3)
>>> v1 == v2 # Deterministic
True
>>> np.array_equal(a1, a2) # Deterministic
True
"""
base = policy_unit_id or os.getenv("POLICY_UNIT_ID", "default")
salt = correlation_id or os.getenv("CORRELATION_ID", "run")
s = _seed_from("fixed", base, salt)

# Set seeds for both random modules
random.seed(s)
np.random.seed(s)

try:
    yield Seeds(py=s, np=s)
finally:
    # Keep deterministic state; caller may reseed per-phase if needed
    pass
```

```
def create_deterministic_rng(seed: int) -> np.random.Generator:
"""
```

```
Create a deterministic NumPy random number generator.
```

```
Use this for local RNG that doesn't affect global state.
```

Args:

```
seed: Integer seed
```

Returns:

```
NumPy Generator instance
```

Examples:

```
>>> rng = create_deterministic_rng(42)
>>> v1 = rng.random()
>>> rng = create_deterministic_rng(42)
>>> v2 = rng.random()
>>> v1 == v2
True
"""
return np.random.default_rng(seed)
```

```
if __name__ == "__main__":
    import doctest
```

```

# Run doctests
print("Running doctests...")
doctest.testmod(verbose=True)

# Integration tests
print("\n" + "="*60)
print("Determinism Integration Tests")
print("="*60)

print("\n1. Testing seed derivation:")
s1 = _seed_from("PU_123", "corr-1")
s2 = _seed_from("PU_123", "corr-1")
s3 = _seed_from("PU_123", "corr-2")
assert s1 == s2
assert s1 != s3
print(f"    ✓ Same inputs → same seed: {s1}")
print(f"    ✓ Different inputs → different seed: {s3}")

print("\n2. Testing deterministic context with random:")
with deterministic("PU_123", "corr-1") as seeds1:
    a = random.random()
    b = random.randint(0, 100)
with deterministic("PU_123", "corr-1") as seeds2:
    c = random.random()
    d = random.randint(0, 100)
assert a == c
assert b == d
print(f"    ✓ Python random is deterministic: {a:.6f}")
print(f"    ✓ Python randint is deterministic: {b}")

print("\n3. Testing deterministic context with numpy:")
with deterministic("PU_123", "corr-1") as seeds:
    arr1 = np.random.rand(3).tolist()
with deterministic("PU_123", "corr-1") as seeds:
    arr2 = np.random.rand(3).tolist()
assert arr1 == arr2
print(f"    ✓ NumPy random is deterministic: {arr1}")

print("\n4. Testing local RNG generator:")
rng1 = create_deterministic_rng(42)
v1 = rng1.random()
rng2 = create_deterministic_rng(42)
v2 = rng2.random()
assert v1 == v2
print(f"    ✓ Local RNG is deterministic: {v1:.6f}")

print("\n5. Testing different correlation IDs produce different results:")
with deterministic("PU_123", "corr-A"):
    val_a = random.random()
with deterministic("PU_123", "corr-B"):
    val_b = random.random()
assert val_a != val_b
print(f"    ✓ Different correlation → different values")
print(f"    corr-A: {val_a:.6f}")
print(f"    corr-B: {val_b:.6f}")

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

```

```

===== FILE: src/saaaaaa/utils/deterministic_execution.py =====
"""

```

#### Deterministic Execution Utilities - Production Grade

```

=====

```

Utilities for ensuring deterministic, reproducible execution across the policy analysis pipeline.

#### Features:

- Deterministic random seed management
- UTC-only timestamp handling
- Structured execution logging
- Side-effect isolation
- Reproducible event ID generation

Author: Policy Analytics Research Unit

Version: 1.0.0

License: Proprietary

"""

```
import hashlib
import logging
import random
import time
import uuid
from collections.abc import Callable, Iterator
from contextlib import contextmanager
from datetime import datetime, timezone
from typing import Any

import numpy as np

from .enhanced_contracts import StructuredLogger, utc_now_iso
from saaaaaa.core.calibration.decorators import calibrated_method
```

```
# =====
# DETERMINISTIC SEED MANAGEMENT
# =====
```

```
class DeterministicSeedManager:
```

```
    """
```

```
    Manages random seeds for deterministic execution.
```

```
    All stochastic operations must use seeds managed by this class to ensure
    reproducibility across runs.
```

```
    Examples:
```

```
    >>> manager = DeterministicSeedManager(base_seed=42)
    >>> with manager.scoped_seed("operation1"):
    ...     value = random.random()
    >>> # Seed is automatically restored after context
    """
```

```
    def __init__(self, base_seed: int = 42) -> None:
```

```
        """
```

```
        Initialize seed manager with base seed.
```

```
        Args:
```

```
        base_seed: Master seed for all derived seeds
```

```
        """
```

```
        self.base_seed = base_seed
```

```
        self._seed_counter = 0
```

```
        self._initialize_seeds(base_seed)
```

```
    @calibrated_method("saaaaaa.utils.deterministic_execution.DeterministicSeedManager._in
_initialize_seeds")
```

```
    def _initialize_seeds(self, seed: int) -> None:
```

```
        """Initialize all random number generators with deterministic seeds."""
```

```
        random.seed(seed)
```

```
        np.random.seed(seed)
```

```
        # For reproducibility, also set hash seed
```

```
        # Note: PYTHONHASHSEED should be set in environment for full determinism
```

```
    @calibrated_method("saaaaaa.utils.deterministic_execution.DeterministicSeedManager.get
_derived_seed")
```

```

def get_derived_seed(self, operation_name: str) -> int:
    """
    Generate a deterministic seed for a specific operation.

    Args:
        operation_name: Unique name for the operation

    Returns:
        Deterministic integer seed derived from operation name and base seed

    Examples:
        >>> manager = DeterministicSeedManager(42)
        >>> seed1 = manager.get_derived_seed("test")
        >>> seed2 = manager.get_derived_seed("test")
        >>> seed1 == seed2 # Deterministic
        True
    """
    # Use cryptographic hash for stable seed derivation
    hash_input = f"{self.base_seed}:{operation_name}".encode()
    hash_digest = hashlib.sha256(hash_input).digest()
    # Convert first 4 bytes to int
    return int.from_bytes(hash_digest[:4], byteorder='big')

@contextmanager
@calibrated_method("saaaaaa.utils.deterministic_execution.DeterministicSeedManager.scoped_seed")
def scoped_seed(self, operation_name: str) -> Iterator[int]:
    """
    Context manager for scoped seed usage.

    Sets seeds for the operation, then restores original state.

    Args:
        operation_name: Unique name for the operation

    Yields:
        Derived seed for this operation

    Examples:
        >>> manager = DeterministicSeedManager(42)
        >>> with manager.scoped_seed("my_operation") as seed:
        ...     result = random.randint(0, 100)
    """
    # Save current state
    random_state = random.getstate()
    np_state = np.random.get_state()

    # Set new seed
    derived_seed = self.get_derived_seed(operation_name)
    self._initialize_seeds(derived_seed)

    try:
        yield derived_seed
    finally:
        # Restore state
        random.setstate(random_state)
        np.random.set_state(np_state)

@calibrated_method("saaaaaa.utils.deterministic_execution.DeterministicSeedManager.get_event_id")
def get_event_id(self, operation_name: str, timestamp_utc: str | None = None) -> str:
    """
    Generate a reproducible event ID for an operation.

    Args:
        operation_name: Operation name
        timestamp_utc: Optional UTC timestamp (ISO-8601); if None, uses current time
    """

```

Returns:

Deterministic event ID based on operation and timestamp

Examples:

```
>>> manager = DeterministicSeedManager(42)
>>> event_id = manager.get_event_id("test", "2024-01-01T00:00:00Z")
>>> len(event_id)
64
"""
ts = timestamp_utc or utc_now_iso()
hash_input = f"{self.base_seed}:{operation_name}:{ts}".encode()
return hashlib.sha256(hash_input).hexdigest()
```

```
# =====
# DETERMINISTIC EXECUTION WRAPPER
# =====
```

class DeterministicExecutor:

```
"""
Wraps functions to ensure deterministic execution with observability.
```

Features:

- Automatic seed management
- Structured logging of execution
- Latency tracking
- Error handling with event IDs

Examples:

```
>>> executor = DeterministicExecutor(base_seed=42, logger_name="test")
>>> @executor.deterministic(operation_name="my_func")
... def my_function(x: int) -> int:
...     return x + random.randint(0, 10)
"""
```

```
def __init__(
    self,
    base_seed: int = 42,
    logger_name: str = "deterministic_executor",
    enable_logging: bool = True
) -> None:
    """
```

Initialize deterministic executor.

Args:

```
    base_seed: Master seed for all operations
    logger_name: Logger name for structured logging
    enable_logging: Whether to enable structured logging
    """
```

```
self.seed_manager = DeterministicSeedManager(base_seed)
self.logger = StructuredLogger(logger_name) if enable_logging else None
self.enable_logging = enable_logging
```

```
def deterministic(
    self,
    operation_name: str,
    log_inputs: bool = False,
    log_outputs: bool = False
) -> Callable:
    """
```

Decorator to make a function deterministic with logging.

Args:

```
    operation_name: Unique name for this operation
    log_inputs: Whether to log input parameters
    log_outputs: Whether to log output values
```

Returns:

```

Decorated function with deterministic execution
"""
def decorator(func: Callable) -> Callable:
    def wrapper(*args: Any, **kwargs: Any) -> Any:
        # Generate correlation and event IDs
        correlation_id = str(uuid.uuid4())
        event_id = self.seed_manager.get_event_id(operation_name)

        # Start timing
        start_time = time.perf_counter()

        # Execute with scoped seed
        try:
            with self.seed_manager.scoped_seed(operation_name) as seed:
                result = func(*args, **kwargs)

            # Calculate latency
            latency_ms = (time.perf_counter() - start_time) * 1000

            # Log success
            if self.enable_logging and self.logger:
                log_data = {
                    "event_id": event_id,
                    "seed": seed,
                    "latency_ms": latency_ms,
                }
                if log_inputs:
                    log_data["inputs"] = str(args)[:100] # Truncate for
                    safety

                if log_outputs:
                    log_data["outputs"] = str(result)[:100]

                self.logger.log_execution(
                    operation=operation_name,
                    correlation_id=correlation_id,
                    success=True,
                    latency_ms=latency_ms,
                    **log_data
                )

            return result

        except Exception as e:
            # Calculate latency even on error
            latency_ms = (time.perf_counter() - start_time) * 1000

            # Log error
            if self.enable_logging and self.logger:
                self.logger.log_execution(
                    operation=operation_name,
                    correlation_id=correlation_id,
                    success=False,
                    latency_ms=latency_ms,
                    event_id=event_id,
                    error=str(e)[:200] # Truncate for safety
                )

            # Re-raise with event ID
            raise RuntimeError(f"[{event_id}] {operation_name} failed: {e}") from
            e

        return wrapper
    return decorator

# =====
# UTC TIMESTAMP UTILITIES
# =====

```

```

def enforce_utc_now() -> datetime:
    """
    Get current UTC datetime.

    Returns:
        Current datetime in UTC timezone

    Examples:
        >>> dt = enforce_utc_now()
        >>> dt.tzinfo is not None
        True
    """
    return datetime.now(timezone.utc)


def parse_utc_timestamp(timestamp_str: str) -> datetime:
    """
    Parse ISO-8601 timestamp and enforce UTC.

    Args:
        timestamp_str: ISO-8601 timestamp string

    Returns:
        Parsed datetime in UTC

    Raises:
        ValueError: If timestamp is not UTC or invalid format

    Examples:
        >>> dt = parse_utc_timestamp("2024-01-01T00:00:00Z")
        >>> dt.year
        2024
    """
    dt = datetime.fromisoformat(timestamp_str.replace('Z', '+00:00'))

    # Enforce UTC
    if dt.tzinfo is None or dt.utcoffset() != timezone.utc.utcoffset(None):
        raise ValueError(f"Timestamp must be UTC: {timestamp_str}")

    return dt


# =====
# SIDE-EFFECT ISOLATION
# =====

@contextmanager
def isolated_execution() -> Iterator[None]:
    """
    Context manager to isolate side effects during execution.

    Current isolation:
    - Prevents print statements (captured and logged as warning)
    - Future: file I/O restrictions, network restrictions

    Yields:
        None

    Examples:
        >>> with isolated_execution():
        ...     # Code here has controlled side effects
        ...     pass
    """
    # For now, minimal isolation - can be extended with more restrictions
    import io
    import sys

```

```

# Capture stdout/stderr to detect violations
old_stdout = sys.stdout
old_stderr = sys.stderr
stdout_capture = io.StringIO()
stderr_capture = io.StringIO()

try:
    sys.stdout = stdout_capture
    sys.stderr = stderr_capture
    yield
finally:
    sys.stdout = old_stdout
    sys.stderr = old_stderr

# Log any captured output as warning (side effect violation)
if stdout_capture.getvalue():
    logging.warning(
        "Side effect detected: stdout captured during isolated execution: %s",
        stdout_capture.getvalue()[:200]
    )
if stderr_capture.getvalue():
    logging.warning(
        "Side effect detected: stderr captured during isolated execution: %s",
        stderr_capture.getvalue()[:200]
    )

# =====
# IN-SCRIPT TESTS
# =====

if __name__ == "__main__":
    import doctest

    # Run doctests
    print("Running doctests...")
    doctest.testmod(verbose=True)

    # Additional tests
    print("\n" + "="*60)
    print("Deterministic Execution Tests")
    print("="*60)

    # Test 1: Seed manager determinism
    print("\n1. Testing seed manager determinism:")
    manager1 = DeterministicSeedManager(42)
    manager2 = DeterministicSeedManager(42)

    seed1_a = manager1.get_derived_seed("test_op")
    seed1_b = manager1.get_derived_seed("test_op")
    seed2_a = manager2.get_derived_seed("test_op")

    assert seed1_a == seed1_b == seed2_a, "Seeds must be deterministic"
    print(f"    ✓ Deterministic seeds: {seed1_a} == {seed1_b} == {seed2_a}")

    # Test 2: Scoped seed restoration
    print("\n2. Testing scoped seed restoration:")
    manager = DeterministicSeedManager(42)

    initial_value = random.random()
    with manager.scoped_seed("temp_operation"):
        _ = random.random() # Different value inside scope
    restored_value = random.random()

    # Reset and check if we can reproduce
    manager._initialize_seeds(42)
    reproduced_value = random.random()

```

```

print(f" ✓ Initial value: {initial_value:.6f}")
print(f" ✓ Reproduced value: {reproduced_value:.6f}")
assert abs(initial_value - reproduced_value) < 1e-10, "Seed restoration failed"
print(" ✓ Seed restoration successful")

```

```

# Test 3: Deterministic executor
print("\n3. Testing deterministic executor:")
executor = DeterministicExecutor(base_seed=42, enable_logging=False)

```

```

@executor.deterministic(operation_name="test_function")
def sample_function(n: int) -> float:
    return sum(random.random() for _ in range(n))

```

```

result1 = sample_function(5)

```

```

# Reset and run again
executor.seed_manager._initialize_seeds(42)
result2 = sample_function(5)

```

```

print(f" ✓ Result 1: {result1:.6f}")
print(f" ✓ Result 2: {result2:.6f}")
assert abs(result1 - result2) < 1e-10, "Deterministic execution failed"
print(" ✓ Deterministic execution verified")

```

```

# Test 4: UTC enforcement
print("\n4. Testing UTC enforcement:")
utc_now = enforce_utc_now()
print(f" ✓ UTC now: {utc_now.isoformat()}")
assert utc_now.tzinfo is not None, "Must have timezone"

```

```

# Test 5: Event ID reproducibility
print("\n5. Testing event ID reproducibility:")
manager = DeterministicSeedManager(42)
event_id1 = manager.get_event_id("operation", "2024-01-01T00:00:00Z")
event_id2 = manager.get_event_id("operation", "2024-01-01T00:00:00Z")
assert event_id1 == event_id2, "Event IDs must be reproducible"
print(f" ✓ Event ID: {event_id1[:16]}...")
print(" ✓ Event ID reproducibility verified")

```

```

print("\n" + "="*60)
print("All tests passed!")
print("="*60)

```

```

===== FILE: src/saaaaaa/utils/domain_errors.py =====
from saaaaaa.core.calibration.decorators import calibrated_method
"""

```

## Domain-Specific Exceptions - Contract Violation Errors

```
=====
```

Provides domain-specific exception hierarchy for contract violations.

### Exception Hierarchy:

```

ContractViolationError (base)
├── DataContractError (data/payload violations)
└── SystemContractError (system/configuration violations)

```

Author: Policy Analytics Research Unit

Version: 1.0.0

License: Proprietary

```

"""

```

```

class ContractViolationError(Exception):
    """

```

Base exception for all contract violations.

Use this as the base class for specific contract violation types.

Examples:

```
>>> try:
...     raise ContractViolationError("Contract violated")
... except ContractViolationError as e:
...     print(f"Caught: {e}")
Caught: Contract violated
"""
pass
```

```
class DataContractError(ContractViolationError):
```

```
"""
```

Exception for data/payload contract violations.

Raised when:

- Payload schema is invalid
- Required fields are missing
- Field values are out of range
- Data integrity checks fail (e.g., digest mismatch)

Examples:

```
>>> try:
...     raise DataContractError("Invalid payload schema")
... except DataContractError as e:
...     print(f"Data error: {e}")
Data error: Invalid payload schema
"""
pass
```

```
class SystemContractError(ContractViolationError):
```

```
"""
```

Exception for system/configuration contract violations.

Raised when:

- System configuration is invalid
- Required resources are unavailable
- Environment preconditions are not met
- Infrastructure failures occur

Examples:

```
>>> try:
...     raise SystemContractError("Configuration missing")
... except SystemContractError as e:
...     print(f"System error: {e}")
System error: Configuration missing
"""
pass
```

```
if __name__ == "__main__":
```

```
    import doctest
```

```
    # Run doctests
```

```
    print("Running doctests...")
```

```
    doctest.testmod(verbose=True)
```

```
    # Integration tests
```

```
    print("\n" + "="*60)
```

```
    print("Domain Exceptions Integration Tests")
```

```
    print("="*60)
```

```
    print("\n1. Testing exception hierarchy:")
```

```
    assert issubclass(DataContractError, ContractViolationError)
```

```
    assert issubclass(SystemContractError, ContractViolationError)
```

```
    print("    ✓ DataContractError inherits from ContractViolationError")
```

```
    print("    ✓ SystemContractError inherits from ContractViolationError")
```

```

print("\n2. Testing exception catching:")
try:
    raise DataContractError("Test data error")
except ContractViolationError as e:
    assert isinstance(e, DataContractError)
    print(" ✓ DataContractError caught as ContractViolationError")

try:
    raise SystemContractError("Test system error")
except ContractViolationError as e:
    assert isinstance(e, SystemContractError)
    print(" ✓ SystemContractError caught as ContractViolationError")

print("\n3. Testing specific exception catching:")
try:
    raise DataContractError("Payload validation failed")
except DataContractError as e:
    assert str(e) == "Payload validation failed"
    print(" ✓ DataContractError caught specifically")

try:
    raise SystemContractError("Config file not found")
except SystemContractError as e:
    assert str(e) == "Config file not found"
    print(" ✓ SystemContractError caught specifically")

print("\n4. Testing error differentiation:")
errors = []

try:
    raise DataContractError("Data issue")
except ContractViolationError as e:
    errors.append(("data", type(e).__name__))

try:
    raise SystemContractError("System issue")
except ContractViolationError as e:
    errors.append(("system", type(e).__name__))

assert errors[0] == ("data", "DataContractError")
assert errors[1] == ("system", "SystemContractError")
print(" ✓ Data and system errors are distinguishable")

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

```

===== FILE: src/saaaaaa/utls/enhanced\_contracts.py =====

"""

Enhanced Contract System with Pydantic - Production Grade

=====

Strict contract definitions with cryptographic verification, deterministic execution guarantees, and comprehensive validation.

Features:

- Static typing with Pydantic BaseModel
- Schema versioning for backward compatibility
- Cryptographic content digests (SHA-256)
- UTC timestamps (ISO-8601)
- Domain-specific exceptions
- Structured JSON logging
- Flow compatibility validation

Author: Policy Analytics Research Unit

Version: 2.0.0

License: Proprietary

"""

```

from __future__ import annotations

import hashlib
import json
import logging
import uuid
from datetime import datetime, timezone
from typing import Any

from pydantic import BaseModel, ConfigDict, Field, field_validator
from saaaaa import get_parameter_loader
from saaaaa.core.calibration.decorators import calibrated_method

# =====
# DOMAIN-SPECIFIC EXCEPTIONS
# =====

class ContractValidationError(Exception):
    """Raised when contract validation fails."""

    def __init__(self, message: str, field: str | None = None, event_id: str | None =
None) -> None:
        self.field = field
        self.event_id = event_id or str(uuid.uuid4())
        super().__init__(f"[{self.event_id}] {message}")

class DataIntegrityError(Exception):
    """Raised when data integrity checks fail (e.g., hash mismatch)."""

    def __init__(self, message: str, expected: str | None = None, got: str | None = None,
event_id: str | None = None) -> None:
        self.expected = expected
        self.got = got
        self.event_id = event_id or str(uuid.uuid4())
        super().__init__(f"[{self.event_id}] {message}")

class SystemConfigError(Exception):
    """Raised when system configuration is invalid."""

    def __init__(self, message: str, config_key: str | None = None, event_id: str | None =
None) -> None:
        self.config_key = config_key
        self.event_id = event_id or str(uuid.uuid4())
        super().__init__(f"[{self.event_id}] {message}")

class FlowCompatibilityError(Exception):
    """Raised when data flow between components is incompatible."""

    def __init__(self, message: str, producer: str | None = None, consumer: str | None =
None, event_id: str | None = None) -> None:
        self.producer = producer
        self.consumer = consumer
        self.event_id = event_id or str(uuid.uuid4())
        super().__init__(f"[{self.event_id}] {message}")

# =====
# UTILITY FUNCTIONS FOR DETERMINISM AND VALIDATION
# =====

def compute_content_digest(content: str | bytes | dict[str, Any]) -> str:
    """
    Compute SHA-256 digest of content in a deterministic way.

```

Args:

content: String, bytes, or dict to hash

Returns:

Hexadecimal SHA-256 digest

Examples:

```
>>> digest = compute_content_digest("test")
>>> len(digest)
64
>>> digest == compute_content_digest("test") # Deterministic
True
"""
if isinstance(content, dict):
    # Sort keys for deterministic JSON
    content_str = json.dumps(content, sort_keys=True, ensure_ascii=True)
    content_bytes = content_str.encode('utf-8')
elif isinstance(content, str):
    content_bytes = content.encode('utf-8')
elif isinstance(content, bytes):
    content_bytes = content
else:
    raise ContractValidationError(
        f"Cannot compute digest for type {type(content).__name__}",
        field="content"
    )

return hashlib.sha256(content_bytes).hexdigest()
```

def utc\_now\_iso() -> str:

"""

Get current UTC timestamp in ISO-8601 format.

Returns:

ISO-8601 timestamp string (UTC timezone)

Examples:

```
>>> ts = utc_now_iso()
>>> 'T' in ts and 'Z' in ts
True
"""
return datetime.now(timezone.utc).isoformat().replace('+00:00', 'Z')
```

```
# =====
# BASE CONTRACT MODEL
# =====
```

class BaseContract(BaseModel):

"""

Base contract model with common fields for all contracts.

All contracts must include:

- schema\_version: Semantic version for contract evolution
- timestamp\_utc: ISO-8601 UTC timestamp
- correlation\_id: UUID for request tracing

"""

```
model_config = ConfigDict(
    frozen=True, # Immutable for safety
    extra='forbid', # Reject unknown fields
    validate_assignment=True,
    str_strip_whitespace=True,
)
```

```
schema_version: str = Field(
    default="2.get_parameter_loader().get("saaaaaa.utils.enhanced_contracts.FlowCompat
```

```

ibilityError.__init__").get("auto_param_L151_19", 0.0)",
    description="Contract schema version (semantic versioning)",
    pattern=r"^\d+\.\d+\.\d+$"
)

timestamp_utc: str = Field(
    default_factory=utc_now_iso,
    description="UTC timestamp in ISO-8601 format"
)

correlation_id: str = Field(
    default_factory=lambda: str(uuid.uuid4()),
    description="UUID for request correlation and tracing"
)

@field_validator('timestamp_utc')
@classmethod
def validate_timestamp(cls, v: str) -> str:
    """Validate timestamp is ISO-8601 format and UTC."""
    try:
        dt = datetime.fromisoformat(v.replace('Z', '+00:00'))
        # Ensure UTC
        if dt.tzinfo is None or dt.utcoffset() != timezone.utc.utcoffset(None):
            raise ValueError("Timestamp must be UTC")
        return v
    except (ValueError, AttributeError) as e:
        raise ContractValidationError(
            f"Invalid ISO-8601 timestamp: {v}",
            field="timestamp_utc"
        ) from e

# =====
# DOCUMENT CONTRACTS - V2
# =====

class DocumentMetadataV2(BaseContract):
    """
    Enhanced document metadata with cryptographic verification.

    Attributes:
        file_path: Absolute path to document
        file_name: Document filename
        num_pages: Number of pages
        file_size_bytes: File size in bytes
        content_digest: SHA-256 hash of file content
        policy_unit_id: Unique identifier for policy unit
        encoding: Character encoding (default: utf-8)
    """

    file_path: str = Field(..., description="Absolute path to document")
    file_name: str = Field(..., description="Document filename", min_length=1)
    num_pages: int = Field(..., description="Number of pages", ge=1)
    file_size_bytes: int = Field(..., description="File size in bytes", ge=0)
    content_digest: str = Field(..., description="SHA-256 hash of content",
    pattern=r"^[a-f0-9]{64}$")
    policy_unit_id: str = Field(..., description="Unique policy unit identifier")
    encoding: str = Field(default="utf-8", description="Character encoding")

    # Optional metadata
    pdf_metadata: dict[str, Any] | None = Field(default=None, description="PDF metadata dictionary")
    author: str | None = Field(default=None, description="Document author")
    title: str | None = Field(default=None, description="Document title")
    creation_date: str | None = Field(default=None, description="Document creation date")

class ProcessedTextV2(BaseContract):

```

```

"""
Enhanced processed text with input/output validation.

Attributes:
    raw_text: Original unprocessed text
    normalized_text: Normalized/cleaned text
    language: Detected language code
    input_digest: SHA-256 of raw_text input
    output_digest: SHA-256 of normalized_text output
    policy_unit_id: Policy unit identifier
    processing_latency_ms: Processing time in milliseconds
"""

raw_text: str = Field(..., description="Original unprocessed text", min_length=1)
normalized_text: str = Field(..., description="Normalized/cleaned text", min_length=1)
language: str = Field(..., description="ISO 639-1 language code",
pattern=r"^[a-z]{2}$")
input_digest: str = Field(..., description="SHA-256 of raw_text",
pattern=r"^[a-f0-9]{64}$")
output_digest: str = Field(..., description="SHA-256 of normalized_text",
pattern=r"^[a-f0-9]{64}$")
policy_unit_id: str = Field(..., description="Policy unit identifier")
processing_latency_ms: float = Field(..., description="Processing latency in ms", ge=0)
et_parameter_loader().get("saaaaaa.utils.enhanced_contracts.FlowCompatibilityError.__init__")
.get("auto_param_L236_89", 0.0))

# Optional fields
sentences: list[str] | None = Field(default=None, description="Sentence segmentation")
sections: list[dict[str, Any]] | None = Field(default=None, description="Document
sections")
payload_size_bytes: int | None = Field(default=None, description="Payload size", ge=0)

@field_validator('input_digest')
@classmethod
def validate_input_digest(cls, v: str, info) -> str:
    """Verify input digest matches raw_text if available."""
    # This is validated post-construction
    return v

# =====
# ANALYSIS CONTRACTS - V2
# =====

class AnalysisInputV2(BaseContract):
    """
    Enhanced analysis input with cryptographic verification.

    Attributes:
        text: Input text to analyze
        document_id: Unique document identifier
        policy_unit_id: Policy unit identifier
        input_digest: SHA-256 of input text
        payload_size_bytes: Size of input payload
    """

    text: str = Field(..., description="Input text to analyze", min_length=1)
    document_id: str = Field(..., description="Unique document identifier")
    policy_unit_id: str = Field(..., description="Policy unit identifier")
    input_digest: str = Field(
        ...,
        description="SHA-256 hash of input text",
        pattern=r"^[a-f0-9]{64}$"
    )
    payload_size_bytes: int = Field(..., description="Payload size in bytes", ge=0)

    # Optional context
    metadata: dict[str, Any] | None = Field(default=None, description="Additional

```

```

metadata")
    context: dict[str, Any] | None = Field(default=None, description="Execution context")
    sentences: list[str] | None = Field(default=None, description="Pre-segmented
sentences")

```

```
@classmethod
```

```
def create_from_text(
```

```
    cls,
```

```
    text: str,
```

```
    document_id: str,
```

```
    policy_unit_id: str,
```

```
    **kwargs: Any
```

```
) -> AnalysisInputV2:
```

```
    """
```

```
    Factory method to create AnalysisInputV2 with auto-computed digest.
```

```
    Args:
```

```
        text: Input text
```

```
        document_id: Document ID
```

```
        policy_unit_id: Policy unit ID
```

```
        **kwargs: Additional optional fields
```

```
    Returns:
```

```
        Validated AnalysisInputV2 instance
```

```
    """
```

```
    input_digest = compute_content_digest(text)
```

```
    payload_size_bytes = len(text.encode('utf-8'))
```

```
    return cls(
```

```
        text=text,
```

```
        document_id=document_id,
```

```
        policy_unit_id=policy_unit_id,
```

```
        input_digest=input_digest,
```

```
        payload_size_bytes=payload_size_bytes,
```

```
        **kwargs
```

```
)
```

```
class AnalysisOutputV2(BaseContract):
```

```
    """
```

```
    Enhanced analysis output with confidence bounds and validation.
```

```
    Attributes:
```

```
        dimension: Analysis dimension
```

```
        category: Result category
```

```
        confidence: Confidence score [get_parameter_loader().get("saaaaaa.
enhanced_contracts.FlowCompatibilityError.__init__").get("auto_param_L322_38", 0.0),
get_parameter_loader().get("saaaaaa.
enhanced_contracts.FlowCompatibilityError.__init__").get("auto_param_L322_43", 1.0)]
```

```
        matches: Evidence matches
```

```
        output_digest: SHA-256 of output content
```

```
        policy_unit_id: Policy unit identifier
```

```
        processing_latency_ms: Processing time in milliseconds
```

```
    """
```

```
    dimension: str = Field(..., description="Analysis dimension", min_length=1)
```

```
    category: str = Field(..., description="Result category", min_length=1)
```

```
    confidence: float = Field(..., description="Confidence score", ge=get_parameter_loader().get("saaaaaa.
enhanced_contracts.FlowCompatibilityError.__init__").get("auto_param_L331_70", 0.0), le=get_parameter_loader().get("saaaaaa.
enhanced_contracts.FlowCompatibilityError.__init__").get("auto_param_L331_78", 1.0))
```

```
    matches: list[str] = Field(..., description="Evidence matches")
```

```
    output_digest: str = Field(..., description="SHA-256 of output",
pattern=r"^[a-f0-9]{64}$")
```

```
    policy_unit_id: str = Field(..., description="Policy unit identifier")
```

```
    processing_latency_ms: float = Field(..., description="Processing latency in ms", ge=get_parameter_loader().get("saaaaaa.
enhanced_contracts.FlowCompatibilityError.__init__").get("auto_param_L335_89", 0.0))
```

```

# Optional fields
positions: list[int] | None = Field(default=None, description="Match positions")
evidence: list[str] | None = Field(default=None, description="Supporting evidence")
warnings: list[str] | None = Field(default=None, description="Validation warnings")
payload_size_bytes: int | None = Field(default=None, description="Output payload
size", ge=0)

@field_validator('confidence')
@classmethod
def validate_confidence_numerical_stability(cls, v: float) -> float:
    """Ensure confidence is numerically stable and within bounds."""
    if not (get_parameter_loader().get("saaaaaa.utils.enhanced_contracts.FlowCompatibi
lityError.__init__").get("auto_param_L347_16", 0.0) <= v <= get_parameter_loader().get("sa
aaaaa.utils.enhanced_contracts.FlowCompatibilityError.__init__").get("auto_param_L347_28",
1.0)):
        raise ContractValidationError(
            f"Confidence must be in [get_parameter_loader().get("saaaaaa.utils.enhance
d_contracts.FlowCompatibilityError.__init__").get("auto_param_L349_41", 0.0), get_paramete
r_loader().get("saaaaaa.utils.enhanced_contracts.FlowCompatibilityError.__init__").get("au
to_param_L349_46", 1.0)], got {v}",
            field="confidence"
        )
    # Round to avoid floating point precision issues
    return round(v, 6)

```

```

# =====
# EXECUTION CONTRACTS - V2
# =====

```

```

class ExecutionContextV2(BaseContract):
    """
    Enhanced execution context with full observability.

    Attributes:
        class_name: Executor class name
        method_name: Method being executed
        document_id: Document identifier
        policy_unit_id: Policy unit identifier
        execution_id: Unique execution identifier
        parent_correlation_id: Parent request correlation ID
    """

```

```

class_name: str = Field(..., description="Executor class name", min_length=1)
method_name: str = Field(..., description="Method being executed", min_length=1)
document_id: str = Field(..., description="Document identifier")
policy_unit_id: str = Field(..., description="Policy unit identifier")
execution_id: str = Field(
    default_factory=lambda: str(uuid.uuid4()),
    description="Unique execution identifier"
)
parent_correlation_id: str | None = Field(
    default=None,
    description="Parent correlation ID for nested calls"
)

```

```

# Optional context
raw_text: str | None = Field(default=None, description="Raw input text")
text: str | None = Field(default=None, description="Processed text")
metadata: dict[str, Any] | None = Field(default=None, description="Metadata")
tables: dict[str, Any] | None = Field(default=None, description="Extracted tables")
sentences: list[str] | None = Field(default=None, description="Sentences")

```

```

# =====
# STRUCTURED LOGGING HELPER
# =====

```

```

class StructuredLogger:
    """
    Structured JSON logger for observability.

    Logs include:
    - correlation_id for tracing
    - latencies per operation
    - payload sizes
    - cryptographic fingerprints
    - NO PII
    """

    def __init__(self, name: str) -> None:
        """Initialize logger with name."""
        self.logger = logging.getLogger(name)
        self.logger.setLevel(logging.INFO)

    def log_contract_validation(
        self,
        contract_type: str,
        correlation_id: str,
        success: bool,
        latency_ms: float,
        payload_size_bytes: int = 0,
        content_digest: str | None = None,
        error: str | None = None
    ) -> None:
        """Log contract validation event."""
        log_entry = {
            "event": "contract_validation",
            "contract_type": contract_type,
            "correlation_id": correlation_id,
            "success": success,
            "latency_ms": round(latency_ms, 3),
            "payload_size_bytes": payload_size_bytes,
            "timestamp_utc": utc_now_iso(),
        }

        if content_digest:
            log_entry["content_digest"] = content_digest

        if error:
            log_entry["error"] = error

        self.logger.info(json.dumps(log_entry, sort_keys=True))

    def log_execution(
        self,
        operation: str,
        correlation_id: str,
        success: bool,
        latency_ms: float,
        **kwargs: Any
    ) -> None:
        """Log execution event with additional context."""
        log_entry = {
            "event": "execution",
            "operation": operation,
            "correlation_id": correlation_id,
            "success": success,
            "latency_ms": round(latency_ms, 3),
            "timestamp_utc": utc_now_iso(),
        }
        log_entry.update(kwargs)

        self.logger.info(json.dumps(log_entry, sort_keys=True))

```

```

# =====
# IN-SCRIPT TESTS
# =====

if __name__ == "__main__":
    import doctest

    # Run doctests
    print("Running doctests...")
    doctest.testmod(verbose=True)

    # Contract validation examples
    print("\n" + "="*60)
    print("Contract Validation Examples")
    print("="*60)

    # Example 1: Document metadata
    print("\n1. DocumentMetadataV2 validation:")
    doc_meta = DocumentMetadataV2(
        file_path="/path/to/document.pdf",
        file_name="document.pdf",
        num_pages=10,
        file_size_bytes=1024000,
        content_digest="a" * 64, # Valid SHA-256 hex
        policy_unit_id="PDM-001"
    )
    print(f" ✓ Valid: correlation_id={doc_meta.correlation_id[:8]}...")

    # Example 2: Analysis input with auto-digest
    print("\n2. AnalysisInputV2 with auto-computed digest:")
    analysis_input = AnalysisInputV2.create_from_text(
        text="Sample policy text for analysis",
        document_id="DOC-123",
        policy_unit_id="PDM-001"
    )
    print(f" ✓ Valid: input_digest={analysis_input.input_digest[:16]}...")
    print(f" ✓ Payload size: {analysis_input.payload_size_bytes} bytes")

    # Example 3: Analysis output with confidence validation
    print("\n3. AnalysisOutputV2 with confidence bounds:")
    analysis_output = AnalysisOutputV2(
        dimension="Dimension1",
        category="CategoryA",
        confidence=get_parameter_loader().get("saaaaaa.utils.enhanced_contracts.Structured
Logger.__init__").get("auto_param_L509_19", 0.85), # Must be in [get_parameter_loader().g
et("saaaaaa.utils.enhanced_contracts.StructuredLogger.__init__").get("auto_param_L509_40",
0.0), get_parameter_loader().get("saaaaaa.utils.enhanced_contracts.StructuredLogger.__ini
t__").get("auto_param_L509_45", 1.0)]
        matches=["evidence1", "evidence2"],
        output_digest="b" * 64,
        policy_unit_id="PDM-001",
        processing_latency_ms=123.456
    )
    print(f" ✓ Valid: confidence={analysis_output.confidence}")

    # Example 4: Structured logging
    print("\n4. Structured logging example:")
    logger = StructuredLogger("test_logger")
    logger.log_contract_validation(
        contract_type="AnalysisInputV2",
        correlation_id=analysis_input.correlation_id,
        success=True,
        latency_ms=5.2,
        payload_size_bytes=analysis_input.payload_size_bytes,
        content_digest=analysis_input.input_digest
    )
    print(" ✓ JSON log emitted to logger")

```

```
# Example 5: Exception handling
print("\n5. Domain-specific exceptions:")
try:
    raise ContractValidationError("Invalid field", field="test_field")
except ContractValidationError as e:
    print(f"    ✓ ContractValidationError: {e}")
    print(f"    ✓ Event ID: {e.event_id}")

print("\n" + "="*60)
print("All validation examples passed!")
print("="*60)
```

```
===== FILE: src/saaaaaa/utils/evidence_registry.py =====
"""Append-only evidence registry with cryptographic hashing.
```

This module implements a small ledger that stores evidence entries produced by analysis components. Each entry links to the previous one through a SHA-256 hash, producing an immutable chain that can be verified for tampering.

```
"""
from __future__ import annotations

import hashlib
import json
from dataclasses import asdict, dataclass
from datetime import datetime
from pathlib import Path
from typing import TYPE_CHECKING, Any
from saaaaaa.core.calibration.decorators import calibrated_method

if TYPE_CHECKING:
    from collections.abc import Iterable

def _canonical_json(payload: dict[str, Any]) -> str:
    """Return a canonical JSON representation with sorted keys."""
    return json.dumps(payload, ensure_ascii=False, sort_keys=True, separators=(",", ":"))

@dataclass(frozen=True)
class EvidenceRecord:
    """Single append-only evidence entry."""

    index: int
    timestamp: str
    method_name: str
    evidence: list[str]
    metadata: dict[str, Any]
    previous_hash: str
    entry_hash: str

    @staticmethod
    def create(
        index: int,
        method_name: str,
        evidence: Iterable[str],
        metadata: dict[str, Any] | None,
        previous_hash: str,
        timestamp: datetime | None = None,
    ) -> EvidenceRecord:
        """Build a new evidence record and compute its hash."""
        ts = (timestamp or datetime.utcnow()).isoformat() + "Z"
        metadata_dict = dict(metadata or {})
        evidence_list = list(evidence)

        payload = {
            "index": index,
            "timestamp": ts,
            "method_name": method_name,
            "evidence": evidence_list,
```

```

        "metadata": metadata_dict,
        "previous_hash": previous_hash,
    }
    digest = hashlib.sha256(_canonical_json(payload).encode("utf-8")).hexdigest()
    return EvidenceRecord(
        index=index,
        timestamp=ts,
        method_name=method_name,
        evidence=evidence_list,
        metadata=metadata_dict,
        previous_hash=previous_hash,
        entry_hash=digest,
    )

class EvidenceRegistry:
    """Append-only registry that persists evidence records to disk."""

    def __init__(self, storage_path: Path | None = None, auto_load: bool = True) -> None:
        self.storage_path = storage_path or Path(".evidence_registry.json")
        self._records: list[EvidenceRecord] = []
        if auto_load and self.storage_path.exists():
            self._records = self._load_records(self.storage_path)

    @property
    @calibrated_method("saaaaaa.utils.evidence_registry.EvidenceRegistry.records")
    def records(self) -> tuple[EvidenceRecord, ...]:
        """Expose records as an immutable tuple."""
        return tuple(self._records)

    def append(
        self,
        method_name: str,
        evidence: Iterable[str],
        metadata: dict[str, Any] | None = None,
        monolith_hash: str | None = None,
    ) -> EvidenceRecord:
        """Append a new evidence record to the registry.

        Args:
            method_name: Name of the method producing evidence
            evidence: Evidence strings
            metadata: Additional metadata dictionary
            monolith_hash: SHA-256 hash of questionnaire_monolith.json (recommended)

        ARCHITECTURAL NOTE: Including monolith_hash ensures evidence is
        traceable to the specific questionnaire version that generated it.
        Use factory.compute_monolith_hash() to generate this value.
        """
        previous_hash = self._records[-1].entry_hash if self._records else "GENESIS"

        # Merge monolith_hash into metadata if provided
        enriched_metadata = dict(metadata or {})
        if monolith_hash is not None:
            enriched_metadata['monolith_hash'] = monolith_hash

        record = EvidenceRecord.create(
            index=len(self._records),
            method_name=method_name,
            evidence=evidence,
            metadata=enriched_metadata,
            previous_hash=previous_hash,
        )
        self._records.append(record)
        return record

# -----
# Persistence
# -----

```

```

@calibrated_method("saaaaaa.utils.evidence_registry.EvidenceRegistry.save")
def save(self) -> None:
    """Persist the registry to disk."""
    payload = [_serialize_record(record) for record in self._records]
    self.storage_path.write_text(
        json.dumps(payload, indent=2, ensure_ascii=False),
        encoding="utf-8",
    )

@calibrated_method("saaaaaa.utils.evidence_registry.EvidenceRegistry._load_records")
def _load_records(self, path: Path) -> list[EvidenceRecord]:
    try:
        data = json.loads(path.read_text(encoding="utf-8"))
    except json.JSONDecodeError as exc:
        raise ValueError(f"Evidence registry at {path} is not valid JSON: {exc}") from exc

    if not isinstance(data, list):
        raise ValueError("Evidence registry payload must be a list")

    records: list[EvidenceRecord] = []
    for index, raw in enumerate(data):
        if not isinstance(raw, dict):
            raise ValueError("Evidence record must be a JSON object")
        expected_index = raw.get("index")
        if expected_index != index:
            raise ValueError(
                f"Evidence record index mismatch at position {index}: found {expected_index}"
            )
        record = EvidenceRecord(
            index=index,
            timestamp=str(raw.get("timestamp")),
            method_name=str(raw.get("method_name")),
            evidence=list(raw.get("evidence", [])),
            metadata=dict(raw.get("metadata", {})),
            previous_hash=str(raw.get("previous_hash")),
            entry_hash=str(raw.get("entry_hash")),
        )
        records.append(record)

    self._assert_chain(records)
    return records

# -----
# Verification utilities
# -----
@calibrated_method("saaaaaa.utils.evidence_registry.EvidenceRegistry.verify")
def verify(self) -> bool:
    """Verify registry integrity by recomputing all hashes."""
    self._assert_chain(self._records)
    return True

@staticmethod
def _assert_chain(records: list[EvidenceRecord]) -> None:
    previous_hash = "GENESIS"
    for expected_index, record in enumerate(records):
        if record.index != expected_index:
            raise ValueError(
                f"Evidence record out of order: expected index {expected_index}, got {record.index}"
            )
        payload = {
            "index": record.index,
            "timestamp": record.timestamp,
            "method_name": record.method_name,
            "evidence": record.evidence,
            "metadata": record.metadata,

```

```

        "previous_hash": previous_hash,
    }
    computed =
hashlib.sha256(_canonical_json(payload).encode("utf-8")).hexdigest()
    if computed != record.entry_hash:
        raise ValueError(
            "Evidence record hash mismatch at index "
            f"{record.index}: expected {computed}, found {record.entry_hash}"
        )
    previous_hash = record.entry_hash

def _serialize_record(record: EvidenceRecord) -> dict[str, Any]:
    payload = asdict(record)
    payload["evidence"] = list(record.evidence)
    payload["metadata"] = dict(record.metadata)
    return payload

__all__ = [
    "EvidenceRecord",
    "EvidenceRegistry",
]

===== FILE: src/saaaaaa/utls/json_contract_loader.py =====
"""Utility helpers to load and validate JSON contract documents."""
from __future__ import annotations

import hashlib
import json
from dataclasses import dataclass
from pathlib import Path
from typing import TYPE_CHECKING, Union
from saaaaaa.core.calibration.decorators import calibrated_method

if TYPE_CHECKING:
    from collections.abc import Iterable, Mapping

PathLike = Union[str, Path]

def _canonical_dump(payload: Mapping[str, object]) -> str:
    return json.dumps(payload, ensure_ascii=False, sort_keys=True, separators=(",", ":"))

@dataclass(frozen=True)
class ContractDocument:
    """Materialized JSON contract with checksum information."""

    path: Path
    payload: dict[str, object]
    checksum: str

@dataclass
class ContractLoadReport:
    """Result of attempting to load multiple contract documents."""

    documents: dict[str, ContractDocument]
    errors: list[str]

    @property

@calibrated_method("saaaaaa.utls.json_contract_loader.ContractLoadReport.is_successful")
def is_successful(self) -> bool:
    return not self.errors

@calibrated_method("saaaaaa.utls.json_contract_loader.ContractLoadReport.summary")
def summary(self) -> str:
    parts = [f"contracts={len(self.documents)}"]
    if self.errors:
        parts.append(f"errors={len(self.errors)}")
    return ", ".join(parts)

```

```

class JSONContractLoader:
    """Load JSON contract files and compute integrity metadata.

    ARCHITECTURAL BOUNDARY: This loader is for generic JSON contracts ONLY.
    It must NOT be used to load questionnaire_monolith.json directly.

    For questionnaire access, use:
    - factory.load_questionnaire() for canonical loading (returns CanonicalQuestionnaire)
    - QuestionnaireResourceProvider for pattern extraction
    """

    def __init__(self, base_path: Path | None = None) -> None:
        self.base_path = base_path or Path(__file__).resolve().parent

    @calibrated_method("saaaaaa.utils.json_contract_loader.JSONContractLoader.load")
    def load(self, paths: Iterable[PathLike]) -> ContractLoadReport:
        documents: dict[str, ContractDocument] = {}
        errors: list[str] = []
        for raw in paths:
            path = self._resolve_path(raw)
            try:
                payload = self._read_payload(path)
            except (FileNotFoundError, json.JSONDecodeError, ValueError) as exc:
                errors.append(f"{path}: {exc}")
            continue

        checksum =
        hashlib.sha256(_canonical_dump(payload).encode("utf-8")).hexdigest()
        documents[str(path)] = ContractDocument(path=path, payload=payload,
        checksum=checksum)
        return ContractLoadReport(documents=documents, errors=errors)

    @calibrated_method("saaaaaa.utils.json_contract_loader.JSONContractLoader.load_directory")
    def load_directory(self, relative_directory: PathLike, pattern: str = "*.json") ->
    ContractLoadReport:
        directory = self._resolve_path(relative_directory)
        if not directory.exists():
            return ContractLoadReport(documents={}, errors=[f"Directory not found:
{directory}"])
        if not directory.is_dir():
            return ContractLoadReport(documents={}, errors=[f"Not a directory:
{directory}"])

        paths = sorted(directory.glob(pattern))
        return self.load(paths)

    # -----
    # Helpers
    # -----

    @calibrated_method("saaaaaa.utils.json_contract_loader.JSONContractLoader._resolve_path")
    def _resolve_path(self, raw: PathLike) -> Path:
        path = Path(raw)
        if not path.is_absolute():
            path = self.base_path / path
        return path

    @staticmethod
    def _read_payload(path: Path) -> dict[str, object]:
        # ARCHITECTURAL GUARD: Block unauthorized questionnaire monolith access
        if path.name == "questionnaire_monolith.json":
            raise ValueError(
                "ARCHITECTURAL VIOLATION: questionnaire_monolith.json must ONLY be "
                "loaded via factory.load_questionnaire() which enforces hash verification.
"
                "Use factory.load_questionnaire() for canonical loading."

```

```

)

text = path.read_text(encoding="utf-8")
data = json.loads(text)
if not isinstance(data, dict):
    raise ValueError("Contract document must be a JSON object")
return data

__all__ = [
    "ContractDocument",
    "ContractLoadReport",
    "JSONContractLoader",
]

===== FILE: src/saaaaaa/utls/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 saaaaaa.core.calibration.decorators import calibrated_method

# Import will be available at runtime
try:
    from .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("saaaaaa.utls.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 = "saaaaaa") -> 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 saaaaaa.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",
    extra={
        "timestamp_utc": envelope_out.timestamp_utc,
        "event_id": envelope_out.event_id,
        "correlation_id": envelope_out.correlation_id,
        "policy_unit_id": envelope_out.policy_unit_id,
        "phase": phase,
        "latency_ms": elapsed_ms,
        "input_bytes": input_bytes,
        "output_bytes": output_bytes,
        "input_digest": getattr(envelope_in, "content_digest", None),
        "output_digest": envelope_out.content_digest,
    },
)

if __name__ == "__main__":
    import doctest
    import time

    # Run doctests
    print("Running doctests...")
    doctest.testmod(verbose=True)

    # Integration tests
    print("\n" + "="*60)
    print("JSON Logger Integration Tests")
    print("="*60)

    print("\n1. Testing JSON formatter:")
    logger = get_json_logger("demo")
    assert logger.level == logging.INFO
    assert len(logger.handlers) > 0
    assert isinstance(logger.handlers[0].formatter, JsonFormatter)

```

```

print(" ✓ Logger configured with JSON formatter")

print("\n2. Testing log output structure:")
# Create a test record
record = logging.LogRecord(
    name="test",
    level=logging.INFO,
    pathname="",
    lineno=0,
    msg="test message",
    args=(),
    exc_info=None,
)
record.event_id = "evt-123"
record.correlation_id = "corr-456"
record.latency_ms = 42

formatter = JsonFormatter()
output = formatter.format(record)
parsed = json.loads(output)

assert parsed["level"] == "INFO"
assert parsed["message"] == "test message"
assert parsed["event_id"] == "evt-123"
assert parsed["correlation_id"] == "corr-456"
assert parsed["latency_ms"] == 42
print(" ✓ 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 .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(" ✓ I/O event logged with correct structure")
    else:
        print(" ✓ I/O event logging executed (output suppressed)")
else:
    print(" ⌘ Skipped (ContractEnvelope not available)")

```

```

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

===== FILE: src/saaaaaa/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 saaaaaa.utils.paths import proj_root
from saaaaaa import get_parameter_loader
from saaaaaa.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
        error_msgs = "\n".join(f" - {err}" for err in validation_errors)
        super().__init__(
            f"Schema validation failed for {file_path}: \n{error_msgs}"
        )

```

```

class MetadataMissingKeyError(MetadataError):
    """Required key missing in metadata"""
    def __init__(self, file_path: str, missing_key: str, context: str = "") -> None:
        self.file_path = file_path
        self.missing_key = missing_key
        self.context = context
        msg = f"Required key '{missing_key}' missing in {file_path}"
        if context:
            msg += f" ({context})"
        super().__init__(msg)

class MetadataLoader:
    """
    Unified metadata loader with strict validation

    Features:
    - Version pinning with semantic versioning
    - SHA-256 checksum verification
    - JSON Schema validation
    - Fail-fast on any violation
    - Structured logging of all errors
    """

    def __init__(self, workspace_root: Path | None = None) -> None:
        self.workspace_root = Path(workspace_root) if workspace_root else proj_root()
        self.schemas_dir = self.workspace_root / "schemas"

        # Loaded schemas cache
        self._schema_cache: dict[str, dict] = {}

    def load_and_validate_metadata(
        self,
        path: Path,
        schema_ref: str | None = None,
        required_version: str | None = None,
        expected_checksum: str | None = None,
        checksum_algorithm: str = "sha256"
    ) -> dict[str, Any]:
        """
        Load and validate metadata file with all safeguards

        Args:
        path: Path to metadata file (JSON or YAML)
        schema_ref: Schema file name (e.g., "rubric.schema.json")
        required_version: Required version string (e.g., "2.get_parameter_loader().get(
('saaaaaa.utils.metadata_loader.MetadataLoader.__init__').get("auto_param_L105_64", 0.0)")
        expected_checksum: Expected SHA-256 checksum (hex)
        checksum_algorithm: Hash algorithm ("sha256", "md5")

        Returns:
        Validated metadata dictionary

        Raises:
        MetadataVersionError: Version mismatch
        MetadataIntegrityError: Checksum mismatch
        MetadataSchemaError: Schema validation failure
        MetadataMissingKeyError: Required key missing
        """

        # 1. Load file
        metadata = self._load_file(path)

        # 2. Version check
        if required_version:
            actual_version = metadata.get("version")
            if not actual_version:
                raise MetadataMissingKeyError(str(path), "version", "version field

```

required")

```
    if actual_version != required_version:
        self._log_error(
            rule_id="VERSION_MISMATCH",
            file_path=str(path),
            expected=required_version,
            actual=actual_version
        )
        raise MetadataVersionError(required_version, actual_version, str(path))

    logger.info(f"✓ Version validated: {path.name} v{actual_version}")

# 3. Checksum verification
if expected_checksum:
    actual_checksum = self._calculate_checksum(metadata, checksum_algorithm)

    if actual_checksum != expected_checksum:
        self._log_error(
            rule_id="CHECKSUM_MISMATCH",
            file_path=str(path),
            expected=expected_checksum,
            actual=actual_checksum
        )
        raise MetadataIntegrityError(str(path), expected_checksum,
actual_checksum)
```

```
    logger.info(f"✓ Checksum validated: {path.name} ({checksum_algorithm})")
```

# 4. Schema validation

```
if schema_ref and JSONSCHEMA_AVAILABLE:
    schema = self._load_schema(schema_ref)
    errors = self._validate_schema(metadata, schema)
```

```
if errors:
    self._log_error(
        rule_id="SCHEMA_VALIDATION_FAILED",
        file_path=str(path),
        errors=errors
    )
    raise MetadataSchemaError(str(path), errors)
```

```
logger.info(f"✓ Schema validated: {path.name}")
```

return metadata

@calibrated\_method("saaaaaa.utils.metadata\_loader.MetadataLoader.\_load\_file")

def \_load\_file(self, path: Path) -> dict[str, Any]:

"""Load JSON or YAML file"""

```
    if not path.exists():
        raise FileNotFoundError(f"Metadata file not found: {path}")
```

try:

```
    with open(path, encoding='utf-8') as f:
        content = f.read()
```

```
    if path.suffix in ['.json']:
        return json.loads(content)
    elif path.suffix in ['.yaml', '.yml']:
        return yaml.safe_load(content)
    else:
        raise ValueError(f"Unsupported file type: {path.suffix}")
```

except (json.JSONDecodeError, yaml.YAMLError) as e:

```
    raise MetadataError(f"Failed to parse {path}: {e}")
```

@calibrated\_method("saaaaaa.utils.metadata\_loader.MetadataLoader.\_calculate\_checksum")

def \_calculate\_checksum(self, metadata: dict[str, Any], algorithm: str = "sha256") ->

```

str:
"""
Calculate reproducible checksum of metadata

Normalization:
- JSON serialization with sorted keys
- UTF-8 encoding
- No whitespace variations
"""

normalized = json.dumps(metadata, sort_keys=True, separators=(',', ':'))

if algorithm == "sha256":
    return hashlib.sha256(normalized.encode('utf-8')).hexdigest()
elif algorithm == "md5":
    return hashlib.md5(normalized.encode('utf-8')).hexdigest()
else:
    raise ValueError(f"Unsupported algorithm: {algorithm}")

@calibrated_method("saaaaaa.utils.metadata_loader.MetadataLoader._load_schema")
def _load_schema(self, schema_ref: str) -> dict[str, Any]:
    """Load JSON Schema from schemas directory"""
    if schema_ref in self._schema_cache:
        return self._schema_cache[schema_ref]

    schema_path = self.schemas_dir / schema_ref

    if not schema_path.exists():
        raise FileNotFoundError(f"Schema not found: {schema_path}")

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

    self._schema_cache[schema_ref] = schema
    return schema

@calibrated_method("saaaaaa.utils.metadata_loader.MetadataLoader._validate_schema")
def _validate_schema(self, metadata: dict[str, Any], schema: dict[str, Any]) -> list:
    """Validate metadata against JSON Schema"""
    if not JSONSCHEMA_AVAILABLE:
        logger.warning("jsonschema not available - skipping schema validation")
        return []

    # Import check for type checker
    import jsonschema as js
    validator = js.Draft7Validator(schema)
    errors = []

    for error in validator.iter_errors(metadata):
        error_path = '.'.join(str(p) for p in error.path) if error.path else 'root'
        errors.append(f"{error_path}: {error.message}")

    return errors

@calibrated_method("saaaaaa.utils.metadata_loader.MetadataLoader._log_error")
def _log_error(self, rule_id: str, file_path: str, **kwargs) -> None:
    """Structured error logging"""
    from datetime import datetime, timezone

    log_entry = {
        "timestamp": datetime.now(timezone.utc).isoformat(),
        "level": "ERROR",
        "rule_id": rule_id,
        "file_path": file_path,
        **kwargs
    }

    logger.error(json.dumps(log_entry, indent=2))

```

```
# REMOVED: load_cuestionario() - LEGACY FUNCTION
# Questionnaire monolith must ONLY be loaded via factory.load_questionnaire_monolith()
# This enforces architectural requirement: Single I/O boundary in factory.py
# See: src/saaaaaa/core/orchestrator/factory.py::load_questionnaire_monolith()
```

```
def load_execution_mapping(
    path: Path | None = None,
    required_version: str = "2.get_parameter_loader().get('saaaaaa.utils.metadata_loader.MetadataLoader._log_error').get('auto_param_L267_31', 0.0)"
) -> dict[str, Any]:
    """
    Load and validate execution_mapping.yaml

    Args:
        path: Path to execution mapping (default: execution_mapping.yaml)
        required_version: Required version

    Returns:
        Validated execution mapping
    """
    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.get_parameter_loader().get('saaaaaa.utils.metadata_loader.MetadataLoader._log_error').get('auto_param_L291_31', 0.0)"
) -> 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/saaaaaa/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 saaaaaa import get_parameter_loader
from saaaaaa.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("saaaaaa.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("saaaaaa.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("saaaaaa.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:
                raise ValueError(f"{param_name}={value} not in allowed set {spec}")

        return True

    @calibrated_method("saaaaaa.utils.method_config_loader.MethodConfigLoader._parse_range")
    def _parse_range(self, spec: str) -> tuple[float, float]:
        """
        Parse range specification like '[get_parameter_loader().get("saaaaaa.utils.method_config_loader.MethodConfigLoader._parse_range").get("auto_param_L135_41", 0.0), get_parameter_loader().get("saaaaaa.utils.method_config_loader.MethodConfigLoader._parse_range").get("auto_param_L135_46", 1.0)], inclusive' or '[100, 10000], integer'.

        Args:
            spec: Range specification string with format "[min, max], modifiers"
            Modifiers can include: inclusive, exclusive, integer

```

Returns:

Tuple of (min\_val, max\_val) as floats

Raises:

ValueError: If spec format is invalid

Note:

The inclusive/exclusive and integer modifiers are parsed but not currently enforced in validation. This maintains compatibility with the current spec while allowing future enhancement.

"""

try:

# Extract bracketed part before any modifiers

bracket\_part = spec.split("]")[0] + "]"

parts = bracket\_part.replace("[", "").replace("]", "").split(",")

min\_val = float(parts[0].strip())

max\_val = float(parts[1].strip())

return min\_val, max\_val

except (IndexError, ValueError) as e:

raise ValueError(f"Invalid range spec: {spec}") from e

@calibrated\_method("saaaaaa.utils.method\_config\_loader.MethodConfigLoader.\_parse\_set")

def \_parse\_set(self, spec: str | list) -> set:

"""

Parse set specification safely.

Args:

spec: Either a list or a string representation of a Python literal

Returns:

Set of allowed values

Raises:

ValueError: If spec cannot be parsed safely

Note:

Uses ast.literal\_eval() for safe parsing of string specs.

Only Python literals (strings, numbers, tuples, lists, dicts,

booleans, None) are supported - no arbitrary code execution.

"""

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/saaaaaa/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 saaaaaa.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."""
    pass
```

```
class UnnormalizedPathError(PathError):
    """Raised when a path is not properly normalized."""
    pass
```

```
# Project root detection - computed once at module load
```

```
def _detect_project_root() -> Path:
```

```
    """
```

```
    Detect the project root directory using filesystem markers.
```

```
    This function uses a multi-strategy approach to locate the project root:
```

1. Primary strategy: Search for pyproject.toml
  - Walks up the directory tree from this file's location
  - Returns the first directory containing pyproject.toml
2. Secondary strategy: Search for src/saaaaaa layout
  - Looks for directories with both src/saaaaaa and setup.py
  - This supports older project structures
3. Fallback strategy: Relative path calculation
  - If no markers found, assumes standard layout (src/saaaaaa/utills)
  - Returns path 3 levels up from this file

```
The function is called once at module load time, and the result is
cached in the PROJECT_ROOT constant.
```

```
Returns:
```

```
    Path: Absolute path to the project root directory
```

```
Raises:
```

```
    No exceptions raised; always returns a path (uses fallback if needed)
```

```
Note:
```

```
    This function is intended for internal use. External code should use
    the PROJECT_ROOT constant instead of calling this directly.
```

```
    """
```

```
# Start from this file's location
```

```
current = Path(__file__).resolve().parent
```

```
# Walk up to find pyproject.toml
```

```

for parent in [current] + list(current.parents):
    if (parent / "pyproject.toml").exists():
        return parent
    if (parent / "src" / "saaaaaa").exists() and (parent / "setup.py").exists():
        return parent

# Fallback: if we can't find it, assume we're in src/saaaaaa/utils
# and go up 3 levels
return current.parent.parent.parent

# Global constants for common directories
PROJECT_ROOT: Final[Path] = _detect_project_root()
SRC_DIR: Final[Path] = PROJECT_ROOT / "src"
DATA_DIR: Final[Path] = PROJECT_ROOT / "data"
TESTS_DIR: Final[Path] = PROJECT_ROOT / "tests"

def proj_root() -> Path:
    """
    Get the project root directory.

    Returns:
        Absolute path to the project root (where pyproject.toml lives)
    """
    return PROJECT_ROOT

def src_dir() -> Path:
    """Get the src directory path."""
    return SRC_DIR

def data_dir() -> Path:
    """
    Get the data directory path.
    Creates it if it doesn't exist.
    """
    DATA_DIR.mkdir(parents=True, exist_ok=True)
    return DATA_DIR

def tmp_dir() -> Path:
    """
    Get a project-specific temporary directory.

    Uses PROJECT_ROOT/tmp to keep temporary files within the workspace
    and avoid polluting system temp directories.

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

def build_dir() -> Path:
    """
    Get the build directory for generated artifacts.

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

```

```
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., "saaaaaa.core")
        *path_parts: Path components within the package

    Returns:
        Path to the resource

    Raises:
        PathNotFoundError: If resource doesn't exist

    Example:
        >>> resources("saaaaaa.core", "config", "default.yaml")
        Path('/path/to/saaaaaa/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}'"
    )
```

# Prohibit writing to source tree unless explicitly allowed

```
if not allow_source_tree and is_within(SRC_DIR, path):
    raise ValueError(
        f"Cannot write to source tree: '{path}'. "
        f"Write to build/, cache/, or reports/ instead. "
        f"If you need to write to source (e.g., code generation), "
        f"set allow_source_tree=True."
    )
```

# Ensure parent directory exists and is writable

```
parent = path.parent
if parent.exists() and not os.access(parent, os.W_OK):
    raise PermissionError(f"Parent directory is not writable: '{parent}'")
```

# Environment variable accessors (typed and safe)

```
def get_env_path(key: str, default: Path | None = None) -> Path | None:
```

```
"""
```

Get a path from environment variable.

Args:

key: Environment variable name  
default: Default value if not set

Returns:

Path from environment or default

```
"""
```

```
value = os.getenv(key)
if value is None:
    return default
return Path(value).resolve()
```

```
def get_workdir() -> Path:
```

```
"""
```

Get the working directory from FLUX\_WORKDIR env var or default to project root.

```
"""
```

```
return get_env_path("FLUX_WORKDIR", PROJECT_ROOT) or PROJECT_ROOT
```

```
def get_tmpdir() -> Path:
```

```
"""
```

Get the temporary directory from FLUX\_TMPDIR env var or default to project tmp.

```
"""
```

```
result = get_env_path("FLUX_TMPDIR", tmp_dir()) or tmp_dir()
```

```

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

```

```

def get_reports_dir() -> Path:

```

```

    """
    Get the reports directory from FLUX_REPORTS env var or default to build/reports.
    """

```

```

    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/saaaaaa/utls/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 saaaaaa.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}")
```

```
    sha256 = hashlib.sha256()
```

```
    with open(file_path, 'rb') as f:
```

```
        # Read in chunks for large files
```

```
        for chunk in iter(lambda: f.read(65536), b''):
```

```
            sha256.update(chunk)
```

```
    return sha256.hexdigest()
```

```
def compute_dict_hash(data: dict[str, Any]) -> str:
```

```
    """Compute SHA-256 hash of a dictionary.
```

```
    The dictionary is serialized with sort_keys=True and ensure_ascii=True
```

```
    to ensure deterministic hashing.
```

```
    Args:
```

```
        data: Dictionary to hash
```

```
    Returns:
```

```
        Hex string of SHA-256 hash
```

```
    """
```

```
json_str = json.dumps(data, sort_keys=True, ensure_ascii=True, separators=(',', ':'))
return hashlib.sha256(json_str.encode('utf-8')).hexdigest()
```

```
def compute_code_signatures(src_root: Path) -> dict[str, str]:
    """Compute SHA-256 hashes of core orchestrator files.
```

Args:

src\_root: Root path to the src/saaaaaa directory

Returns:

Dictionary mapping filename to SHA-256 hash

Raises:

FileNotFoundError: If any required file is missing  
"""

```
core_files = {
    'core.py': src_root / 'core' / 'orchestrator' / 'core.py',
    'executors.py': src_root / 'core' / 'orchestrator' / 'executors.py',
    'factory.py': src_root / 'core' / 'orchestrator' / 'factory.py',
}

signatures = {}
for name, path in core_files.items():
    if not path.exists():
        raise FileNotFoundError(f"Required core file missing: {path}")
    signatures[name] = compute_file_hash(path)

return signatures
```

```
def verify_success_conditions(
    phase_results: list[Any],
    abort_active: bool,
    output_dir: Path,
) -> 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)

# Write proof.hash
proof_hash_path = output_dir / 'proof.hash'
with open(proof_hash_path, 'w', encoding='utf-8') as f:
    f.write(proof_hash)

return proof_json_path, proof_hash_path

```

```

def collect_artifacts_manifest(output_dir: Path) -> dict[str, str]:
    """Collect hashes of all artifacts in output directory.

```

Args:

output\_dir: Directory containing artifacts

Returns:

Dictionary mapping relative path to SHA-256 hash

"""

```

manifest = {}

```

```

# Find all artifacts (JSON, MD, logs)

```

```

patterns = ['*.json', '*.md', '*.log', '*.txt']

```

```

for pattern in patterns:

```

```

    for artifact_path in output_dir.rglob(pattern):

```

```

        # Skip proof files themselves

```

```

        if artifact_path.name in ('proof.json', 'proof.hash'):

```

```

            continue

```

```

        try:

```

```

            rel_path = artifact_path.relative_to(output_dir)

```

```

            manifest[str(rel_path)] = compute_file_hash(artifact_path)

```

```

        except (OSError, PermissionError, ValueError):

```

```

            # Skip files we can't read or hash

```

```

            pass

```

```

return manifest

```

```

def verify_proof(proof_json_path: Path, proof_hash_path: Path) -> tuple[bool, str]:
    """Verify that a proof.json file matches its proof.hash.

```

This allows anyone to verify that the proof hasn't been tampered with.

Args:

proof\_json\_path: Path to proof.json

proof\_hash\_path: Path to proof.hash

Returns:

Tuple of (valid: bool, message: str)

"""

```

try:

```

```

    # Read proof.json

```

```

    with open(proof_json_path, encoding='utf-8') as f:

```

```

        proof_dict = json.load(f)

```

```

    # 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, "✓ Proof verified: hash matches"
else:
    return False, f"✗ Proof verification failed: hash mismatch\n Expected:
{stored_hash}\n Got: {computed_hash}"

except Exception as e:
    return False, f"✗ Proof verification error: {e}"

```

===== FILE: src/saaaaaa/utls/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 saaaaaa import get_parameter_loader
from saaaaaa.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"""

```

```

        self.recording_path = recording_path or Path(".qmcm_recording.json")

```

```

        self.calls: list[dict[str, Any]] = []

```

```

        self.enabled = True

```

```

    def record_call(

```

```

        self,

```

```

        method_name: str,

```

```

        input_types: dict[str, str],

```

```

        output_type: str,

```

```

        execution_status: str = "success",

```

```

        execution_time_ms: float = get_parameter_loader().get("saaaaaa.utls.qmcm_hook

```

```

s.QMCMRecorder.__init__").get("auto_param_L45_39", 0.0),

```

```

        monolith_hash: str | None = None

```

```

    ) -> None:

```

```

        """

```

```

        Record a method call

```

Args:

method\_name: Name of the method called  
input\_types: Dictionary mapping parameter names to type names  
output\_type: Type name of the return value  
execution\_status: 'success', 'error', or 'skipped'  
execution\_time\_ms: Execution time in milliseconds  
monolith\_hash: SHA-256 hash of questionnaire\_monolith.json (recommended)

ARCHITECTURAL NOTE: Including monolith\_hash ties each method call to the specific questionnaire version, enabling reproducibility. Use factory.compute\_monolith\_hash() to generate this value.

"""

if not self.enabled:  
 return

```
call_record = {  
    "timestamp": datetime.now().isoformat(),  
    "method_name": method_name,  
    "input_types": input_types,  
    "output_type": output_type,  
    "execution_status": execution_status,  
    "execution_time_ms": round(execution_time_ms, 2)  
}
```

# Include monolith\_hash if provided  
if monolith\_hash is not None:  
 call\_record["monolith\_hash"] = monolith\_hash

self.calls.append(call\_record)  
logger.debug(f"QMCM recorded: {method\_name}")

@calibrated\_method("saaaaaa.utils.qmcm\_hooks.QMCMRecorder.get\_statistics")

def get\_statistics(self) -> dict[str, Any]:

"""

Get recording statistics

Returns summary of method call patterns without data content  
"""

if not self.calls:

```
    return {  
        "total_calls": 0,  
        "unique_methods": 0,  
        "method_frequency": {},  
        "success_rate": get_parameter_loader().get("saaaaaa.utils.qmcm_hooks.QMCMR  
ecorder.get_statistics").get("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 get\_paramet

```
er_loader().get("saaaaaa.utils.qmcm_hooks.QMCMRecorder.get_statistics").get("auto_param_L1
16_79", 0.0),
    "most_called_method": most_called
}
```

```
@calibrated_method("saaaaaa.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("saaaaaa.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)
```

```
self.calls = recording_data.get("calls", [])
logger.info(f"QMCM recording loaded: {len(self.calls)} calls")
```

```
@calibrated_method("saaaaaa.utils.qmcm_hooks.QMCMRecorder.clear_recording")
```

```
def clear_recording(self) -> None:
    """Clear all recorded calls"""
    self.calls = []
    logger.info("QMCM recording cleared")
```

```
@calibrated_method("saaaaaa.utils.qmcm_hooks.QMCMRecorder.enable")
```

```
def enable(self) -> None:
    """Enable recording"""
    self.enabled = True
```

```
@calibrated_method("saaaaaa.utils.qmcm_hooks.QMCMRecorder.disable")
```

```
def disable(self) -> None:
    """Disable recording"""
    self.enabled = False
```

```
# Global recorder instance
```

```
_global_recorder: QMCMRecorder | None = None
```

```
def get_global_recorder() -> QMCMRecorder:
```

```
    """Get or create global QMCM recorder"""
    global _global_recorder
    if _global_recorder is None:
        _global_recorder = QMCMRecorder()
    return _global_recorder
```

```
def qmcm_record(method=None, *, monolith_hash: str | None = None):
```

```
    """
    Decorator to record method calls in QMCM
```

```
Usage:
```

```
    @qmcm_record
    @calibrated_method("saaaaaa.utils.qmcm_hooks.QMCMRecorder.my_method")
```

```

def my_method(self, arg1: str, arg2: int) -> dict:
    return {"result": "data"}

# With monolith hash (recommended for questionnaire-dependent methods)
@qmcm_record(monolith_hash=compute_monolith_hash(questionnaire))

@calibrated_method("saaaaaa.utils.qmcm_hooks.QMCMRecorder.my_questionnaire_method")
def my_questionnaire_method(self, question_id: str) -> dict:
    return {"result": "data"}
"""
def decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        recorder = get_global_recorder()

        import time
        start_time = time.time()

        try:
            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/saaaaaa/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 saaaaaa.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/saaaaaa/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 saaaaaa import get_parameter_loader
from saaaaaa.core.calibration.decorators import calibrated_method

if TYPE_CHECKING:
    from collections.abc import Mapping
    from pathlib import Path

logger = logging.getLogger(__name__)

# =====
# SCHEMA SHAPE TRACKING
# =====

class SchemaShape(TypedDict):
    """Shape of a data payload."""

    keys: set[str]
    types: dict[str, str]
    sample_values: dict[str, Any]
    timestamp: str

    @dataclass
    class SchemaStats:
        """Statistics about schema shape over time."""

        key_frequency: Counter[str] = field(default_factory=Counter)
        type_by_key: dict[str, Counter[str]] = field(
            default_factory=lambda: defaultdict(Counter)
        )
        new_keys: set[str] = field(default_factory=set)
        missing_keys: set[str] = field(default_factory=set)
        total_samples: int = 0
        last_updated: datetime | None = None

class SchemaDriftDetector:
    """
    Detects schema drift by sampling payloads and tracking shape changes.

    Usage:
        detector = SchemaDriftDetector(sample_rate=0.05)

```

```

# In your API/pipeline
if detector.should_sample():
    detector.record_payload(data, source="api_input")

# Check for drift
alerts = detector.get_alerts()
"""

def __init__(
    self,
    *,
    sample_rate: float = 0.05,
    baseline_path: Path | None = None,
    alert_threshold: float = 0.1,
) -> None:
    """
    Initialize schema drift detector.

    Args:
        sample_rate: Percentage of payloads to sample (0.01 = 1%, 0.05 = 5%)
        baseline_path: Path to baseline schema file
        alert_threshold: Threshold for drift alert (% of samples with drift)
    """
    self.sample_rate = sample_rate
    self.baseline_path = baseline_path
    self.alert_threshold = alert_threshold

    # Tracking state
    self.stats_by_source: dict[str, SchemaStats] = defaultdict(SchemaStats)
    self.baseline_schema: dict[str, SchemaShape] = {}

    # Load baseline if provided
    if baseline_path and baseline_path.exists():
        self._load_baseline()

@calibrated_method("saaaaaa.utils.schema_monitor.SchemaDriftDetector.should_sample")
def should_sample(self) -> bool:
    """Decide whether to sample this payload (probabilistic)."""
    return random.random() < self.sample_rate

def record_payload(
    self,
    payload: Mapping[str, Any],
    *,
    source: str,
    timestamp: datetime | None = None,
) -> None:
    """
    Record a payload for schema tracking.

    Args:
        payload: Data payload to analyze
        source: Source identifier (e.g., "api_input", "document_loader")
        timestamp: Optional timestamp, defaults to now
    """
    if timestamp is None:
        timestamp = datetime.utcnow()

    stats = self.stats_by_source[source]

    # Extract shape
    keys = set(payload.keys())
    types = {k: type(v).__name__ for k, v in payload.items()}

    # Update statistics
    stats.total_samples += 1
    stats.last_updated = timestamp

```

```

for key in keys:
    stats.key_frequency[key] += 1
    stats.type_by_key[key][types[key]] += 1

# Detect new keys (compared to baseline)
if source in self.baseline_schema:
    baseline_keys = self.baseline_schema[source]["keys"]
    new_keys = keys - baseline_keys
    if new_keys:
        stats.new_keys.update(new_keys)
        logger.warning(
            f"SCHEMA_DRIFT[source={source}]: New keys detected: {new_keys}"
        )

    missing_keys = baseline_keys - keys
    if missing_keys:
        stats.missing_keys.update(missing_keys)
        logger.warning(
            f"SCHEMA_DRIFT[source={source}]: Missing keys detected:
{missing_keys}"
        )

@calibrated_method("saaaaaa.utils.schema_monitor.SchemaDriftDetector.get_alerts")
def get_alerts(self, *, source: str | None = None) -> list[dict[str, Any]]:
    """
    Get schema drift alerts.

    Args:
        source: Optional source filter

    Returns:
        List of alert dicts
    """
    alerts: list[dict[str, Any]] = []

    sources = [source] if source else list(self.stats_by_source.keys())

    for src in sources:
        stats = self.stats_by_source[src]

        if stats.new_keys:
            alerts.append({
                "level": "WARNING",
                "source": src,
                "type": "NEW_KEYS",
                "keys": list(stats.new_keys),
                "timestamp": stats.last_updated.isoformat() if stats.last_updated else
None,
            })

        if stats.missing_keys:
            alerts.append({
                "level": "CRITICAL",
                "source": src,
                "type": "MISSING_KEYS",
                "keys": list(stats.missing_keys),
                "timestamp": stats.last_updated.isoformat() if stats.last_updated else
None,
            })

    # Check for type inconsistencies
    for key, type_counts in stats.type_by_key.items():
        if len(type_counts) > 1:
            # Multiple types seen for same key
            dominant_type = type_counts.most_common(1)[0][0]
            other_types = [t for t in type_counts if t != dominant_type]

```

```

        alerts.append({
            "level": "WARNING",
            "source": src,
            "type": "TYPE_INCONSISTENCY",
            "key": key,
            "expected_type": dominant_type,
            "observed_types": other_types,
            "timestamp": stats.last_updated.isoformat() if stats.last_updated
else None,
        })

    return alerts

@calibrated_method("saaaaaa.utils.schema_monitor.SchemaDriftDetector.save_baseline")
def save_baseline(self, output_path: Path) -> None:
    """
    Save current schema shapes as baseline.

    Args:
        output_path: Path to save baseline JSON
    """
    baseline: dict[str, dict[str, Any]] = {}

    for source, stats in self.stats_by_source.items():
        # Get most common keys (present in >50% of samples)
        threshold = stats.total_samples * get_parameter_loader().get("saaaaaa.utils.schema_monitor.SchemaDriftDetector.save_baseline").get("auto_param_L215_46", 0.5)
        common_keys = {
            key for key, count in stats.key_frequency.items()
            if count >= threshold
        }

        # Get dominant type for each key
        types = {
            key: type_counts.most_common(1)[0][0]
            for key, type_counts in stats.type_by_key.items()
        }

        baseline[source] = {
            "keys": list(common_keys),
            "types": types,
            "timestamp": datetime.utcnow().isoformat(),
        }

    output_path.write_text(json.dumps(baseline, indent=2))
    logger.info(f"Saved schema baseline to {output_path}")

@calibrated_method("saaaaaa.utils.schema_monitor.SchemaDriftDetector._load_baseline")
def _load_baseline(self) -> None:
    """Load baseline schema from file."""
    if not self.baseline_path:
        return

    try:
        data = json.loads(self.baseline_path.read_text())

        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("saaaaaa.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:
        msg = f"VALIDATION_ERROR[source={source}]: Missing required keys: {missing}"
        if strict:
            raise ValueError(msg)
        else:
            logger.warning(msg)

    # Check types
    for key, expected_type in expected_types.items():
        if key in payload:
            actual_type = type(payload[key]).__name__
            if actual_type != expected_type:
                msg = (
                    f"VALIDATION_ERROR[source={source}, key={key}]: "
                    f"Expected type {expected_type}, got {actual_type}"
                )
                if strict:
                    raise TypeError(msg)
                else:
                    logger.warning(msg)

@calibrated_method("saaaaaa.utils.schema_monitor.PayloadValidator._load_schemas")
def _load_schemas(self) -> None:
    """Load schema definitions from file."""
    if not self.schema_path:
        return

    try:
        self.schemas = json.loads(self.schema_path.read_text())
        logger.info(f"Loaded schemas from {self.schema_path}")
    except Exception as e:
        logger.error(f"Failed to load schemas: {e}")

# =====
# GLOBAL INSTANCE (optional convenience)
# =====

```

```

# Singleton detector for application-wide use
_global_detector: SchemaDriftDetector | None = None

def get_detector() -> SchemaDriftDetector:
    """Get or create global schema drift detector."""
    global _global_detector
    if _global_detector is None:
        _global_detector = SchemaDriftDetector(sample_rate=get_parameter_loader().get("saa
aaaa.utils.schema_monitor.PayloadValidator._load_schemas").get("auto_param_L400_59",
0.05))
    return _global_detector

===== FILE: src/saaaaaa/utils/seed_factory.py =====
"""

Deterministic Seed Factory
Generates reproducible seeds for all stochastic operations
"""

import hashlib
import hmac
import random
from typing import Any
from saaaaaa.core.calibration.decorators import calibrated_method

try:
    import numpy as np
    NUMPY_AVAILABLE = True
except ImportError:
    NUMPY_AVAILABLE = False
    np = None # type: ignore

class SeedFactory:
    """
    Factory for generating deterministic seeds

    Ensures:
    - Reproducibility: Same inputs → same seed
    - Uniqueness: Different contexts → different seeds
    - Cryptographic quality: HMAC-SHA256 derivation
    """

    # Fixed salt for seed derivation (should be configured per deployment)
    DEFAULT_SALT = b"PDM_EVALUATOR_V2_DETERMINISTIC_SEED_2025"

    def __init__(self, fixed_salt: bytes | None = None) -> None:
        self.salt = fixed_salt or self.DEFAULT_SALT

    def create_deterministic_seed(
        self,
        correlation_id: str,
        file_checksums: dict[str, str] | None = None,
        context: dict[str, Any] | None = None
    ) -> int:
        """
        Generate deterministic seed from correlation ID and context

        Args:
            correlation_id: Unique workflow instance identifier
            file_checksums: Dict of {filename: sha256_checksum}
            context: Additional context (question_id, policy_area, etc.)

        Returns:
            32-bit integer seed (0 to 2^32-1)

        Example:
            >>> factory = SeedFactory()
            >>> seed1 = factory.create_deterministic_seed("run-001", {"data.json":
"abc123"})

```

```

    >>> seed2 = factory.create_deterministic_seed("run-001", {"data.json":
"abc123"})
    >>> assert seed1 == seed2 # Reproducible
    """

    # 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("saaaaaa.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"
```