```python
        Returns:
            List of SeedRecord objects with generation history

        Useful for debugging non-determinism issues.
        """
        return list(self._audit_log)

    def clear_cache(self) -> None:
        """Clear seed cache (useful for testing or isolation)."""
        self._seed_cache.clear()
        logger.debug("Seed cache cleared")

    def get_seeds_for_context(
        self,
        policy_unit_id: str,
        correlation_id: str
    ) -> dict[str, int]:
        """
        Get all standard seeds for an execution context.

        Args:
            policy_unit_id: Unique identifier for the policy document/unit
            correlation_id: Unique identifier for this execution context

        Returns:
            Dictionary mapping component names to seeds

        Components:
            - numpy: NumPy RNG initialization
            - python: Python random module seeding
            - quantum: Quantum optimizer initialization
            - neuromorphic: Neuromorphic controller initialization
            - meta_learner: Meta-learner strategy selection
        """
        components = ["numpy", "python", "quantum", "neuromorphic", "meta_learner"]
        return {
            component: self.get_seed(policy_unit_id, correlation_id, component)
            for component in components
        }

    def get_manifest_entry(
        self,
        policy_unit_id: str | None = None,
        correlation_id: str | None = None
    ) -> dict:
        """
        Get manifest entry for verification manifest.

        Args:
            policy_unit_id: Optional filter by policy_unit_id
            correlation_id: Optional filter by correlation_id

        Returns:
            Dictionary suitable for inclusion in verification_manifest.json
        """
        # Filter audit log if criteria provided
        if policy_unit_id or correlation_id:
            filtered_log = [
                record for record in self._audit_log
                if (not policy_unit_id or record.policy_unit_id == policy_unit_id)
                and (not correlation_id or record.correlation_id == correlation_id)
            ]
        else:
            filtered_log = self._audit_log

        # Use first record for base info (they should all have same context)
        base_record = filtered_log[0] if filtered_log else None
```

```python
        manifest = {
            "seed_version": SEED_VERSION,
            "seeds_generated": len(filtered_log),
        }

        if base_record:
            manifest["policy_unit_id"] = base_record.policy_unit_id
            manifest["correlation_id"] = base_record.correlation_id

            # Include seed breakdown by component
            manifest["seeds_by_component"] = {
                record.component: record.seed
                for record in filtered_log
            }

        return manifest


# Global registry instance (singleton pattern)
_global_registry: SeedRegistry | None = None


def get_global_seed_registry() -> SeedRegistry:
    """
    Get or create the global seed registry instance.

    Returns:
        Global SeedRegistry singleton
    """
    global _global_registry
    if _global_registry is None:
        _global_registry = SeedRegistry()
    return _global_registry


def reset_global_seed_registry() -> None:
    """Reset the global seed registry (useful for testing)."""
    global _global_registry
    _global_registry = None
```

===== FILE: src/saaaaaa/core/orchestrator/settings.py =====

```python
"""
Centralized settings module for SAAAAAA orchestrator.
This module loads configuration from environment variables and .env file.
Only the orchestrator should read from this module - core modules should not import this.
"""

import os
from pathlib import Path
from typing import Final

from dotenv import load_dotenv

# Load environment variables from .env file
# Look for .env in the repository root
REPO_ROOT: Final[Path] = Path(__file__).parent.parent
ENV_FILE: Final[Path] = REPO_ROOT / ".env"

if ENV_FILE.exists():
    load_dotenv(ENV_FILE)

def _get_int(key: str, default: int) -> int:
    """Safely get an integer from environment variables."""
    value = os.getenv(key)
    if value is None:
        return default
    try:
        return int(value)
```

```python
    except (ValueError, TypeError):
        return default

def _get_bool(key: str, default: str) -> bool:
    """Safely get a boolean from environment variables."""
    return os.getenv(key, default).lower() == "true"

class Settings:
    """Application settings loaded from environment variables."""

    # Application Settings
    APP_ENV: str = os.getenv("APP_ENV", "development")
    DEBUG: bool = _get_bool("DEBUG", "false")
    LOG_LEVEL: str = os.getenv("LOG_LEVEL", "INFO")

    # API Configuration
    API_HOST: str = os.getenv("API_HOST", "0.0.0.0")
    API_PORT: int = _get_int("API_PORT", 5000)
    API_SECRET_KEY: str = os.getenv("API_SECRET_KEY", "dev-secret-key")

    # Database Configuration
    DB_HOST: str = os.getenv("DB_HOST", "localhost")
    DB_PORT: int = _get_int("DB_PORT", 5432)
    DB_NAME: str = os.getenv("DB_NAME", "saaaaaa")
    DB_USER: str = os.getenv("DB_USER", "saaaaaa_user")
    DB_PASSWORD: str = os.getenv("DB_PASSWORD", "")

    # Redis Configuration
    REDIS_HOST: str = os.getenv("REDIS_HOST", "localhost")
    REDIS_PORT: int = _get_int("REDIS_PORT", 6379)
    REDIS_DB: int = _get_int("REDIS_DB", 0)

    # Authentication
    JWT_SECRET_KEY: str = os.getenv("JWT_SECRET_KEY", "dev-jwt-secret")
    JWT_ALGORITHM: str = os.getenv("JWT_ALGORITHM", "HS256")
    JWT_EXPIRATION_HOURS: int = _get_int("JWT_EXPIRATION_HOURS", 24)

    # External Services
    OPENAI_API_KEY: str = os.getenv("OPENAI_API_KEY", "")
    ANTHROPIC_API_KEY: str = os.getenv("ANTHROPIC_API_KEY", "")

    # Processing Configuration
    MAX_WORKERS: int = _get_int("MAX_WORKERS", 4)
    BATCH_SIZE: int = _get_int("BATCH_SIZE", 100)
    TIMEOUT_SECONDS: int = _get_int("TIMEOUT_SECONDS", 300)

    # Feature Flags
    ENABLE_CACHING: bool = _get_bool("ENABLE_CACHING", "true")
    ENABLE_MONITORING: bool = _get_bool("ENABLE_MONITORING", "false")
    ENABLE_RATE_LIMITING: bool = _get_bool("ENABLE_RATE_LIMITING", "true")

# Global settings instance
settings = Settings()
```

===== FILE: src/saaaaaa/core/orchestrator/signal_aliasing.py =====

```python
"""Signal Aliasing Module - Soft-alias pattern for PA07-PA10 fingerprint canonicalization.

This module implements the soft-alias pattern to prevent duplicate fingerprints
and ensure proper cache invalidation for policy areas PA07-PA10.

Key Features:
- Canonical fingerprint computation from signal content (not static strings)
- Backward-compatible alias mapping for legacy fingerprints
- Cache invalidation support via content-based hashing
- Merkle tree integrity for PA07-PA10

SOTA Requirements:
- Prevents silent degradation from static fingerprints
```

```
- Enables observability for PA coverage gaps
- Supports intelligent fallback fusion
"""

from __future__ import annotations

import hashlib
import json
from typing import TYPE_CHECKING, Any

if TYPE_CHECKING:
    from .signals import SignalPack

try:
    import blake3
    BLAKE3_AVAILABLE = True
except ImportError:
    BLAKE3_AVAILABLE = False

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


def resolve_fingerprint_alias(fingerprint: str, legacy_aliases: dict[str, str]) -> str:
    """
    Resolve legacy fingerprint to canonical policy area using a provided alias map.

    Args:
        fingerprint: Fingerprint to resolve (may be legacy or canonical).
        legacy_aliases: A dictionary mapping legacy fingerprints to canonical IDs.

    Returns:
        Canonical policy area ID (PA01-PA10) or original fingerprint.

    Example:
        >>> aliases = {"pa07_v1_land_territory": "PA07"}
        >>> resolve_fingerprint_alias("pa07_v1_land_territory", aliases)
        'PA07'
        >>> resolve_fingerprint_alias("abc123...", aliases)
        'abc123...'
    """
    return legacy_aliases.get(fingerprint, fingerprint)


def build_fingerprint_index(
    signal_packs: dict[str, SignalPack]
) -> dict[str, str]:
    """
    Build fingerprint index mapping canonical fingerprints to policy areas.

    This creates a reverse index for cache lookups:
    - canonical_fingerprint -> policy_area_id
    - Supports both legacy and content-based fingerprints

    Args:
        signal_packs: Dict mapping policy_area_id to SignalPack

    Returns:
        Dict mapping canonical_fingerprint to policy_area_id

    Example:
        >>> packs = build_all_signal_packs()
        >>> index = build_fingerprint_index(packs)
        >>> print(f"Index size: {len(index)}")
```

```python
    """
    fingerprint_index = {}

    for policy_area_id, signal_pack in signal_packs.items():
        # Compute canonical fingerprint
        canonical_fp = canonicalize_signal_fingerprint(signal_pack)

        # Map canonical fingerprint to policy area
        fingerprint_index[canonical_fp] = policy_area_id

        # Also map legacy fingerprint for backward compatibility
        legacy_fp = signal_pack.source_fingerprint
        if legacy_fp and legacy_fp != canonical_fp:
            fingerprint_index[legacy_fp] = policy_area_id

    logger.info(
        "fingerprint_index_built",
        index_size=len(fingerprint_index),
        policy_areas=len(signal_packs),
    )

    return fingerprint_index


def validate_fingerprint_uniqueness(
    signal_packs: dict[str, SignalPack]
) -> dict[str, Any]:
    """
    Validate that all fingerprints are unique (no duplicates).

    This is a quality gate to prevent fingerprint collisions that would
    break cache invalidation and Merkle tree integrity.

    Args:
        signal_packs: Dict mapping policy_area_id to SignalPack

    Returns:
        Validation result with:
        - is_valid: bool
        - duplicates: list of duplicate fingerprints
        - collisions: dict mapping fingerprint to list of policy areas

    Example:
        >>> packs = build_all_signal_packs()
        >>> result = validate_fingerprint_uniqueness(packs)
        >>> assert result["is_valid"], "Fingerprint collision detected!"
    """
    fingerprint_to_pas = {}

    for policy_area_id, signal_pack in signal_packs.items():
        canonical_fp = canonicalize_signal_fingerprint(signal_pack)

        if canonical_fp not in fingerprint_to_pas:
            fingerprint_to_pas[canonical_fp] = []

        fingerprint_to_pas[canonical_fp].append(policy_area_id)

    # Find duplicates
    duplicates = {
        fp: pas
        for fp, pas in fingerprint_to_pas.items()
        if len(pas) > 1
    }

    is_valid = len(duplicates) == 0

    result = {
        "is_valid": is_valid,
```

```
        "total_fingerprints": len(fingerprint_to_pas),
        "unique_fingerprints": len([pas for pas in fingerprint_to_pas.values() if len(pas)
 == 1]),
        "duplicates": list(duplicates.keys()),
        "collisions": duplicates,
    }

    if not is_valid:
        logger.error(
            "fingerprint_collision_detected",
            duplicates=duplicates,
        )
    else:
        logger.info(
            "fingerprint_uniqueness_validated",
            total_fingerprints=result["total_fingerprints"],
        )

    return result


def upgrade_legacy_fingerprints(
    signal_packs: dict[str, SignalPack]
) -> dict[str, SignalPack]:
    """
    Upgrade legacy fingerprints to canonical content-based fingerprints.

    This is a migration helper for PA07-PA10 to transition from static
    fingerprints to content-based fingerprints.

    Args:
        signal_packs: Dict mapping policy_area_id to SignalPack

    Returns:
        Updated signal_packs with canonical fingerprints

    Example:
        >>> packs = build_all_signal_packs()
        >>> upgraded_packs = upgrade_legacy_fingerprints(packs)
        >>> for pa, pack in upgraded_packs.items():
        >>>     print(f"{pa}: {pack.source_fingerprint}")
    """
    upgraded_packs = {}

    for policy_area_id, signal_pack in signal_packs.items():
        # Compute canonical fingerprint
        canonical_fp = canonicalize_signal_fingerprint(signal_pack)

        # Update source_fingerprint to canonical
        signal_pack.source_fingerprint = canonical_fp

        # Add migration metadata
        if "migration" not in signal_pack.metadata:
            signal_pack.metadata["migration"] = {}

        signal_pack.metadata["migration"]["upgraded_to_canonical"] = True
        signal_pack.metadata["migration"]["canonical_fingerprint"] = canonical_fp

        upgraded_packs[policy_area_id] = signal_pack

    logger.info(
        "legacy_fingerprints_upgraded",
        upgraded_count=len(upgraded_packs),
    )

    return upgraded_packs
```

===== FILE: src/saaaaaa/core/orchestrator/signal_cache_invalidation.py =====

```python
"""Signal Cache Invalidation Module - Content-based cache invalidation for PA signals.

This module implements cache invalidation strategies based on canonical fingerprints,
ensuring that cache entries are invalidated when signal content changes.

Key Features:
- Content-based cache keys (fingerprint-derived)
- TTL-based expiration with grace periods
- Merkle tree validation for cache integrity
- Cache warming for high-traffic PAs
- Invalidation audit trail

SOTA Requirements:
- Prevents stale cache serving from static fingerprints
- Ensures data integrity across PA01-PA10
- Supports soft-alias pattern for PA07-PA10
"""

from __future__ import annotations

import time
from dataclasses import dataclass, field
from typing import TYPE_CHECKING, Any

if TYPE_CHECKING:
    from .signals import SignalPack

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


@dataclass
class CacheEntry:
    """Cache entry for SignalPack.

    Attributes:
        key: Cache key (canonical fingerprint)
        policy_area_id: Policy area identifier
        signal_pack: Cached SignalPack object
        created_at: Creation timestamp (Unix epoch)
        expires_at: Expiration timestamp (Unix epoch)
        access_count: Number of cache hits
        last_accessed: Last access timestamp (Unix epoch)
        metadata: Additional metadata
    """
    key: str
    policy_area_id: str
    signal_pack: SignalPack
    created_at: float
    expires_at: float
    access_count: int = 0
    last_accessed: float = 0.0
    metadata: dict[str, Any] = field(default_factory=dict)

    @property
    def is_expired(self) -> bool:
        """Check if cache entry is expired."""
        return time.time() >= self.expires_at

    @property
    def ttl_remaining(self) -> float:
        """Get remaining TTL in seconds."""
        return max(0.0, self.expires_at - time.time())
```

```
    @property
    def age_seconds(self) -> float:
        """Get age of cache entry in seconds."""
        return time.time() - self.created_at


@dataclass
class CacheInvalidationEvent:
    """Event record for cache invalidation.

    Attributes:
        event_type: Type of invalidation event
        policy_area_id: Affected policy area
        old_fingerprint: Previous fingerprint
        new_fingerprint: New fingerprint (if applicable)
        timestamp: Event timestamp (Unix epoch)
        reason: Human-readable reason
        metadata: Additional metadata
    """
    event_type: str
    policy_area_id: str
    old_fingerprint: str | None
    new_fingerprint: str | None
    timestamp: float
    reason: str
    metadata: dict[str, Any] = field(default_factory=dict)


class SignalPackCache:
    """In-memory cache for SignalPacks with content-based invalidation.

    This implements a simple in-memory cache with:
    - Content-based cache keys (canonical fingerprints)
    - TTL-based expiration
    - Access tracking
    - Invalidation audit trail
    """

    def __init__(self, max_size: int = 100):
        """
        Initialize SignalPackCache.

        Args:
            max_size: Maximum cache size (LRU eviction)
        """
        self.max_size = max_size
        self._cache: dict[str, CacheEntry] = {}
        self._invalidation_log: list[CacheInvalidationEvent] = []

    def get(self, key: str) -> SignalPack | None:
        """
        Get SignalPack from cache.

        Args:
            key: Cache key (canonical fingerprint)

        Returns:
            Cached SignalPack or None if not found/expired

        Example:
            >>> cache = SignalPackCache()
            >>> pack = cache.get("abc123...")
        """
        entry = self._cache.get(key)

        if entry is None:
            logger.debug("cache_miss", key=key[:8])
            return None
```

```
        # Check expiration
        if entry.is_expired:
            logger.debug("cache_expired", key=key[:8], age=entry.age_seconds)
            self._invalidate_entry(key, "expired")
            return None

        # Update access tracking
        entry.access_count += 1
        entry.last_accessed = time.time()

        logger.debug(
            "cache_hit",
            key=key[:8],
            policy_area=entry.policy_area_id,
            access_count=entry.access_count,
        )

        return entry.signal_pack

    def put(
        self,
        key: str,
        policy_area_id: str,
        signal_pack: SignalPack,
        ttl_seconds: float | None = None,
    ) -> None:
        """
        Put SignalPack into cache.

        Args:
            key: Cache key (canonical fingerprint)
            policy_area_id: Policy area identifier
            signal_pack: SignalPack to cache
            ttl_seconds: TTL in seconds (uses signal_pack.ttl_s if None)

        Example:
            >>> cache = SignalPackCache()
            >>> cache.put("abc123...", "PA07", pack)
        """
        # Determine TTL
        if ttl_seconds is None:
            ttl_seconds = signal_pack.ttl_s or 86400.0  # Default 24 hours

        # Check cache size and evict if necessary
        if len(self._cache) >= self.max_size and key not in self._cache:
            self._evict_lru()

        # Create cache entry
        now = time.time()
        entry = CacheEntry(
            key=key,
            policy_area_id=policy_area_id,
            signal_pack=signal_pack,
            created_at=now,
            expires_at=now + ttl_seconds,
            last_accessed=now,
            metadata={
                "version": signal_pack.version,
                "fingerprint": signal_pack.source_fingerprint,
            },
        )

        self._cache[key] = entry

        logger.debug(
            "cache_put",
            key=key[:8],
```

```python
        policy_area=policy_area_id,
        ttl_seconds=ttl_seconds,
    )

def invalidate(self, key: str, reason: str = "manual") -> bool:
    """
    Invalidate cache entry.

    Args:
        key: Cache key to invalidate
        reason: Reason for invalidation

    Returns:
        True if entry was invalidated, False if not found

    Example:
        >>> cache.invalidate("abc123...", "content_changed")
    """
    return self._invalidate_entry(key, reason)

def invalidate_by_policy_area(
    self,
    policy_area_id: str,
    reason: str = "manual",
) -> int:
    """
    Invalidate all cache entries for a policy area.

    Args:
        policy_area_id: Policy area identifier
        reason: Reason for invalidation

    Returns:
        Number of entries invalidated

    Example:
        >>> count = cache.invalidate_by_policy_area("PA07", "signal_updated")
    """
    keys_to_invalidate = [
        key for key, entry in self._cache.items()
        if entry.policy_area_id == policy_area_id
    ]

    for key in keys_to_invalidate:
        self._invalidate_entry(key, reason)

    logger.info(
        "policy_area_invalidated",
        policy_area=policy_area_id,
        invalidated_count=len(keys_to_invalidate),
        reason=reason,
    )

    return len(keys_to_invalidate)

def invalidate_all(self, reason: str = "manual") -> int:
    """
    Invalidate all cache entries.

    Args:
        reason: Reason for invalidation

    Returns:
        Number of entries invalidated

    Example:
        >>> count = cache.invalidate_all("system_restart")
    """
```

```python
        keys = list(self._cache.keys())
        for key in keys:
            self._invalidate_entry(key, reason)

        logger.info(
            "cache_invalidated_all",
            invalidated_count=len(keys),
            reason=reason,
        )

        return len(keys)

    def warm_cache(
        self,
        signal_packs: dict[str, SignalPack],
    ) -> int:
        """
        Warm cache with signal packs.

        Args:
            signal_packs: Dict mapping policy_area_id to SignalPack

        Returns:
            Number of entries warmed

        Example:
            >>> packs = build_all_signal_packs()
            >>> cache.warm_cache(packs)
        """
        # Import here to avoid circular dependency
        from .signal_aliasing import canonicalize_signal_fingerprint

        warmed_count = 0

        for policy_area_id, signal_pack in signal_packs.items():
            # Compute canonical fingerprint
            canonical_fp = canonicalize_signal_fingerprint(signal_pack)

            # Put in cache
            self.put(canonical_fp, policy_area_id, signal_pack)
            warmed_count += 1

        logger.info(
            "cache_warmed",
            warmed_count=warmed_count,
        )

        return warmed_count

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

        Returns:
            Cache statistics dict

        Example:
            >>> stats = cache.get_stats()
            >>> print(f"Size: {stats['size']}, Hit rate: {stats['hit_rate']:.2%}")
        """
        total_accesses = sum(entry.access_count for entry in self._cache.values())
        expired_count = sum(1 for entry in self._cache.values() if entry.is_expired)

        # Compute hit rate (rough estimate)
        invalidation_count = len(self._invalidation_log)
        hit_rate = total_accesses / max(total_accesses + invalidation_count, 1)

        return {
```

```python
            "size": len(self._cache),
            "max_size": self.max_size,
            "total_accesses": total_accesses,
            "expired_count": expired_count,
            "invalidation_count": invalidation_count,
            "hit_rate": hit_rate,
        }

    def _invalidate_entry(self, key: str, reason: str) -> bool:
        """Internal method to invalidate cache entry."""
        entry = self._cache.pop(key, None)

        if entry is None:
            return False

        # Log invalidation event
        event = CacheInvalidationEvent(
            event_type="invalidation",
            policy_area_id=entry.policy_area_id,
            old_fingerprint=entry.metadata.get("fingerprint"),
            new_fingerprint=None,
            timestamp=time.time(),
            reason=reason,
            metadata={
                "age_seconds": entry.age_seconds,
                "access_count": entry.access_count,
            },
        )

        self._invalidation_log.append(event)

        logger.debug(
            "cache_invalidated",
            key=key[:8],
            policy_area=entry.policy_area_id,
            reason=reason,
            age=entry.age_seconds,
        )

        return True

    def _evict_lru(self) -> None:
        """Evict least recently used cache entry."""
        if not self._cache:
            return

        # Find LRU entry
        lru_key = min(
            self._cache.keys(),
            key=lambda k: self._cache[k].last_accessed,
        )

        self._invalidate_entry(lru_key, "lru_eviction")


def build_cache_key(policy_area_id: str, signal_pack: SignalPack) -> str:
    """
    Build cache key from SignalPack using canonical fingerprint.

    This uses the soft-alias pattern to ensure cache keys are content-based.

    Args:
        policy_area_id: Policy area identifier
        signal_pack: SignalPack to compute key for

    Returns:
        Cache key (canonical fingerprint)
```

```python
    Example:
        >>> pack = build_signal_pack_from_monolith("PA07")
        >>> key = build_cache_key("PA07", pack)
        >>> print(f"Cache key: {key}")
    """
    # Import here to avoid circular dependency
    from .signal_aliasing import canonicalize_signal_fingerprint

    canonical_fp = canonicalize_signal_fingerprint(signal_pack)

    logger.debug(
        "cache_key_built",
        policy_area=policy_area_id,
        canonical_fp=canonical_fp[:8],
    )

    return canonical_fp


def validate_cache_integrity(
    cache: SignalPackCache,
    signal_packs: dict[str, SignalPack],
) -> dict[str, Any]:
    """
    Validate cache integrity against current signal packs.

    This checks:
    - All cached entries have valid fingerprints
    - Cached content matches current signal packs
    - No stale entries exist

    Args:
        cache: SignalPackCache to validate
        signal_packs: Dict mapping policy_area_id to SignalPack

    Returns:
        Validation result dict

    Example:
        >>> cache = SignalPackCache()
        >>> cache.warm_cache(packs)
        >>> result = validate_cache_integrity(cache, packs)
        >>> assert result["is_valid"], "Cache integrity violation detected!"
    """
    # Import here to avoid circular dependency
    from .signal_aliasing import canonicalize_signal_fingerprint

    stale_entries = []
    mismatched_entries = []

    for policy_area_id, signal_pack in signal_packs.items():
        # Compute current canonical fingerprint
        canonical_fp = canonicalize_signal_fingerprint(signal_pack)

        # Check if cached
        cached_pack = cache.get(canonical_fp)

        if cached_pack is None:
            continue  # Not cached (OK)

        # Compute cached fingerprint
        cached_fp = canonicalize_signal_fingerprint(cached_pack)

        # Check fingerprint match
        if cached_fp != canonical_fp:
            mismatched_entries.append({
                "policy_area": policy_area_id,
                "expected_fp": canonical_fp[:8],
```

```python
                "actual_fp": cached_fp[:8],
            })

    is_valid = len(stale_entries) == 0 and len(mismatched_entries) == 0

    result = {
        "is_valid": is_valid,
        "stale_entries": stale_entries,
        "mismatched_entries": mismatched_entries,
        "cache_stats": cache.get_stats(),
    }

    if is_valid:
        logger.info("cache_integrity_validated", cache_size=cache.get_stats()["size"])
    else:
        logger.error(
            "cache_integrity_violation",
            stale_count=len(stale_entries),
            mismatch_count=len(mismatched_entries),
        )

    return result


def create_global_cache() -> SignalPackCache:
    """
    Create global SignalPackCache instance.

    This is a convenience factory for creating a cache with sensible defaults.

    Returns:
        SignalPackCache instance

    Example:
        >>> cache = create_global_cache()
        >>> packs = build_all_signal_packs()
        >>> cache.warm_cache(packs)
    """
    cache = SignalPackCache(max_size=100)

    logger.info("global_cache_created", max_size=cache.max_size)

    return cache
```

===== FILE: src/saaaaaa/core/orchestrator/signal_calibration_gate.py =====

```python
"""Signal Calibration Gate Module - Hard quality gates for SOTA requirements.

This module implements hard calibration gates to prevent silent degradation
in signal quality, coverage, and threshold calibration.

Key Features:
- Hard quality thresholds (fail-fast on violations)
- Calibration drift detection (threshold consistency)
- Coverage completeness checks (all PA01-PA10 present)
- Fingerprint uniqueness validation
- Temporal freshness gates (TTL bounds)

SOTA Requirements:
- Prevents silent degradation from misconfigured signals
- Enforces minimum quality bar for production
- Provides actionable error messages for violations
"""

from __future__ import annotations

from dataclasses import dataclass, field
from enum import Enum
from typing import TYPE_CHECKING, Any
```

```python
if TYPE_CHECKING:
    from .signals import SignalPack
    from .signal_quality_metrics import SignalQualityMetrics

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


class GateSeverity(str, Enum):
    """Gate violation severity levels."""
    ERROR = "ERROR"  # Blocks deployment
    WARNING = "WARNING"  # Logged but doesn't block
    INFO = "INFO"  # Informational only


@dataclass
class GateViolation:
    """Container for gate violation details.

    Attributes:
        gate_name: Name of violated gate
        severity: Violation severity
        policy_area_id: Affected policy area (if applicable)
        message: Human-readable error message
        actual_value: Actual measured value
        expected_value: Expected/threshold value
        remediation: Suggested fix
    """
    gate_name: str
    severity: GateSeverity
    policy_area_id: str | None
    message: str
    actual_value: Any
    expected_value: Any
    remediation: str


@dataclass
class CalibrationGateResult:
    """Result of calibration gate validation.

    Attributes:
        passed: Whether all gates passed
        violations: List of gate violations
        summary: Summary statistics
    """
    passed: bool
    violations: list[GateViolation] = field(default_factory=list)
    summary: dict[str, Any] = field(default_factory=dict)

    @property
    def has_errors(self) -> bool:
        """Check if any ERROR-level violations exist."""
        return any(v.severity == GateSeverity.ERROR for v in self.violations)

    @property
    def has_warnings(self) -> bool:
        """Check if any WARNING-level violations exist."""
        return any(v.severity == GateSeverity.WARNING for v in self.violations)

    def get_violations_by_severity(
        self,
        severity: GateSeverity
```

```
    ) -> list[GateViolation]:
        """Get violations filtered by severity."""
        return [v for v in self.violations if v.severity == severity]


@dataclass
class CalibrationGateConfig:
    """Configuration for calibration gates.

    Attributes:
        min_patterns_per_pa: Minimum patterns required per PA
        min_indicators_per_pa: Minimum indicators required per PA
        min_entities_per_pa: Minimum entities required per PA
        min_confidence_threshold: Minimum allowed confidence threshold
        max_confidence_threshold: Maximum allowed confidence threshold
        min_evidence_threshold: Minimum allowed evidence threshold
        max_threshold_drift: Maximum allowed drift between PA thresholds
        min_ttl_hours: Minimum TTL in hours
        max_ttl_hours: Maximum TTL in hours
        require_all_pas: Whether all PA01-PA10 must be present
        require_unique_fingerprints: Whether fingerprints must be unique
    """
    min_patterns_per_pa: int = 10
    min_indicators_per_pa: int = 2
    min_entities_per_pa: int = 2
    min_confidence_threshold: float = 0.70
    max_confidence_threshold: float = 0.95
    min_evidence_threshold: float = 0.65
    max_threshold_drift: float = 0.15
    min_ttl_hours: float = 1.0
    max_ttl_hours: float = 48.0
    require_all_pas: bool = True
    require_unique_fingerprints: bool = True


def validate_pattern_coverage_gate(
    metrics_by_pa: dict[str, SignalQualityMetrics],
    config: CalibrationGateConfig,
) -> list[GateViolation]:
    """
    Validate pattern coverage gate.

    Checks:
    - Each PA has minimum pattern count
    - Each PA has minimum indicator count
    - Each PA has minimum entity count

    Args:
        metrics_by_pa: Dict mapping policy_area_id to SignalQualityMetrics
        config: Calibration gate configuration

    Returns:
        List of gate violations
    """
    violations = []

    for pa, metrics in metrics_by_pa.items():
        # Pattern count
        if metrics.pattern_count < config.min_patterns_per_pa:
            violations.append(GateViolation(
                gate_name="pattern_coverage",
                severity=GateSeverity.ERROR,
                policy_area_id=pa,
                message=f"Insufficient patterns in {pa}",
                actual_value=metrics.pattern_count,
                expected_value=config.min_patterns_per_pa,
                remediation="Extract more patterns from questionnaire or enable fusion",
            ))
```

```
        # Indicator count
        if metrics.indicator_count < config.min_indicators_per_pa:
            violations.append(GateViolation(
                gate_name="indicator_coverage",
                severity=GateSeverity.WARNING,
                policy_area_id=pa,
                message=f"Insufficient indicators in {pa}",
                actual_value=metrics.indicator_count,
                expected_value=config.min_indicators_per_pa,
                remediation="Review INDICADOR patterns in questionnaire",
            ))

        # Entity count
        if metrics.entity_count < config.min_entities_per_pa:
            violations.append(GateViolation(
                gate_name="entity_coverage",
                severity=GateSeverity.WARNING,
                policy_area_id=pa,
                message=f"Insufficient entities in {pa}",
                actual_value=metrics.entity_count,
                expected_value=config.min_entities_per_pa,
                remediation="Review FUENTE_OFICIAL patterns in questionnaire",
            ))

    return violations


def validate_threshold_calibration_gate(
    metrics_by_pa: dict[str, SignalQualityMetrics],
    config: CalibrationGateConfig,
) -> list[GateViolation]:
    """
    Validate threshold calibration gate.

    Checks:
    - Confidence thresholds within bounds
    - Evidence thresholds within bounds
    - Threshold drift across PAs is acceptable

    Args:
        metrics_by_pa: Dict mapping policy_area_id to SignalQualityMetrics
        config: Calibration gate configuration

    Returns:
        List of gate violations
    """
    violations = []

    confidence_thresholds = []
    evidence_thresholds = []

    for pa, metrics in metrics_by_pa.items():
        # Confidence threshold bounds
        if metrics.threshold_min_confidence < config.min_confidence_threshold:
            violations.append(GateViolation(
                gate_name="confidence_threshold_too_low",
                severity=GateSeverity.ERROR,
                policy_area_id=pa,
                message=f"Confidence threshold too low in {pa}",
                actual_value=metrics.threshold_min_confidence,
                expected_value=config.min_confidence_threshold,
                remediation="Increase min_confidence threshold in signal pack",
            ))

        if metrics.threshold_min_confidence > config.max_confidence_threshold:
            violations.append(GateViolation(
                gate_name="confidence_threshold_too_high",
```

```
                severity=GateSeverity.WARNING,
                policy_area_id=pa,
                message=f"Confidence threshold too high in {pa}",
                actual_value=metrics.threshold_min_confidence,
                expected_value=config.max_confidence_threshold,
                remediation="Decrease min_confidence threshold to improve recall",
            ))

        # Evidence threshold bounds
        if metrics.threshold_min_evidence < config.min_evidence_threshold:
            violations.append(GateViolation(
                gate_name="evidence_threshold_too_low",
                severity=GateSeverity.ERROR,
                policy_area_id=pa,
                message=f"Evidence threshold too low in {pa}",
                actual_value=metrics.threshold_min_evidence,
                expected_value=config.min_evidence_threshold,
                remediation="Increase min_evidence threshold in signal pack",
            ))

        confidence_thresholds.append(metrics.threshold_min_confidence)
        evidence_thresholds.append(metrics.threshold_min_evidence)

    # Threshold drift check
    if confidence_thresholds:
        max_confidence = max(confidence_thresholds)
        min_confidence = min(confidence_thresholds)
        confidence_drift = max_confidence - min_confidence

        if confidence_drift > config.max_threshold_drift:
            violations.append(GateViolation(
                gate_name="threshold_drift_excessive",
                severity=GateSeverity.WARNING,
                policy_area_id=None,
                message="Excessive threshold drift across policy areas",
                actual_value=confidence_drift,
                expected_value=config.max_threshold_drift,
                remediation="Recalibrate thresholds for consistency",
            ))

    return violations


def validate_completeness_gate(
    signal_packs: dict[str, SignalPack],
    config: CalibrationGateConfig,
) -> list[GateViolation]:
    """
    Validate completeness gate.

    Checks:
    - All PA01-PA10 are present
    - No missing policy areas

    Args:
        signal_packs: Dict mapping policy_area_id to SignalPack
        config: Calibration gate configuration

    Returns:
        List of gate violations
    """
    violations = []

    if config.require_all_pas:
        expected_pas = {f"PA{i:02d}" for i in range(1, 11)}
        actual_pas = set(signal_packs.keys())
        missing_pas = expected_pas - actual_pas
```

```
        if missing_pas:
            violations.append(GateViolation(
                gate_name="completeness_missing_pas",
                severity=GateSeverity.ERROR,
                policy_area_id=None,
                message="Missing policy areas",
                actual_value=sorted(actual_pas),
                expected_value=sorted(expected_pas),
                remediation=f"Add missing policy areas: {sorted(missing_pas)}",
            ))

    return violations


def validate_fingerprint_uniqueness_gate(
    signal_packs: dict[str, SignalPack],
    config: CalibrationGateConfig,
) -> list[GateViolation]:
    """
    Validate fingerprint uniqueness gate.

    Checks:
    - All fingerprints are unique
    - No duplicate fingerprints

    Args:
        signal_packs: Dict mapping policy_area_id to SignalPack
        config: Calibration gate configuration

    Returns:
        List of gate violations
    """
    violations = []

    if config.require_unique_fingerprints:
        # Import here to avoid circular dependency
        from .signal_aliasing import validate_fingerprint_uniqueness

        result = validate_fingerprint_uniqueness(signal_packs)

        if not result["is_valid"]:
            for fingerprint, pas in result["collisions"].items():
                violations.append(GateViolation(
                    gate_name="fingerprint_collision",
                    severity=GateSeverity.ERROR,
                    policy_area_id=None,
                    message=f"Duplicate fingerprint across policy areas",
                    actual_value=pas,
                    expected_value="unique fingerprints",
                    remediation=f"Use soft-alias pattern to resolve collision: {pas}",
                ))

    return violations


def validate_temporal_freshness_gate(
    metrics_by_pa: dict[str, SignalQualityMetrics],
    config: CalibrationGateConfig,
) -> list[GateViolation]:
    """
    Validate temporal freshness gate.

    Checks:
    - TTL within bounds
    - Temporal bounds set (valid_from/valid_to)

    Args:
        metrics_by_pa: Dict mapping policy_area_id to SignalQualityMetrics
```

```
        config: Calibration gate configuration

    Returns:
        List of gate violations
    """
    violations = []

    for pa, metrics in metrics_by_pa.items():
        # TTL bounds
        if metrics.ttl_hours < config.min_ttl_hours:
            violations.append(GateViolation(
                gate_name="ttl_too_short",
                severity=GateSeverity.WARNING,
                policy_area_id=pa,
                message=f"TTL too short in {pa}",
                actual_value=metrics.ttl_hours,
                expected_value=config.min_ttl_hours,
                remediation="Increase TTL to reduce cache churn",
            ))

        if metrics.ttl_hours > config.max_ttl_hours:
            violations.append(GateViolation(
                gate_name="ttl_too_long",
                severity=GateSeverity.WARNING,
                policy_area_id=pa,
                message=f"TTL too long in {pa}",
                actual_value=metrics.ttl_hours,
                expected_value=config.max_ttl_hours,
                remediation="Decrease TTL to ensure freshness",
            ))

        # Temporal bounds
        if not metrics.has_temporal_bounds:
            violations.append(GateViolation(
                gate_name="missing_temporal_bounds",
                severity=GateSeverity.INFO,
                policy_area_id=pa,
                message=f"Missing temporal bounds in {pa}",
                actual_value=None,
                expected_value="valid_from/valid_to",
                remediation="Set valid_from/valid_to for temporal tracking",
            ))

    return violations


def run_calibration_gates(
    signal_packs: dict[str, SignalPack],
    metrics_by_pa: dict[str, SignalQualityMetrics],
    config: CalibrationGateConfig | None = None,
) -> CalibrationGateResult:
    """
    Run all calibration gates and return comprehensive result.

    This is the main entry point for calibration gate validation.

    Args:
        signal_packs: Dict mapping policy_area_id to SignalPack
        metrics_by_pa: Dict mapping policy_area_id to SignalQualityMetrics
        config: Calibration gate configuration (uses default if None)

    Returns:
        CalibrationGateResult with validation results

    Example:
        >>> packs = build_all_signal_packs()
        >>> metrics = {pa: compute_signal_quality_metrics(pack, pa) for pa, pack in
packs.items()}
```

```
        >>> result = run_calibration_gates(packs, metrics)
        >>> if not result.passed:
        >>>     for violation in result.get_violations_by_severity(GateSeverity.ERROR):
        >>>         print(f"ERROR: {violation.message}")
        >>>     raise ValueError("Calibration gates failed")
    """
    if config is None:
        config = CalibrationGateConfig()

    all_violations: list[GateViolation] = []

    # Run all gates
    all_violations.extend(validate_pattern_coverage_gate(metrics_by_pa, config))
    all_violations.extend(validate_threshold_calibration_gate(metrics_by_pa, config))
    all_violations.extend(validate_completeness_gate(signal_packs, config))
    all_violations.extend(validate_fingerprint_uniqueness_gate(signal_packs, config))
    all_violations.extend(validate_temporal_freshness_gate(metrics_by_pa, config))

    # Classify violations by severity
    errors = [v for v in all_violations if v.severity == GateSeverity.ERROR]
    warnings = [v for v in all_violations if v.severity == GateSeverity.WARNING]
    infos = [v for v in all_violations if v.severity == GateSeverity.INFO]

    # Gates pass only if no errors
    passed = len(errors) == 0

    summary = {
        "total_violations": len(all_violations),
        "errors": len(errors),
        "warnings": len(warnings),
        "infos": len(infos),
        "gates_passed": passed,
        "gates_run": 5,  # Number of gate functions
    }

    result = CalibrationGateResult(
        passed=passed,
        violations=all_violations,
        summary=summary,
    )

    if passed:
        logger.info(
            "calibration_gates_passed",
            total_violations=len(all_violations),
            warnings=len(warnings),
            infos=len(infos),
        )
    else:
        logger.error(
            "calibration_gates_failed",
            total_violations=len(all_violations),
            errors=len(errors),
            warnings=len(warnings),
        )

    return result


def generate_gate_report(result: CalibrationGateResult) -> str:
    """
    Generate human-readable report for calibration gate results.

    Args:
        result: CalibrationGateResult to report

    Returns:
        Formatted report string
```

```python
    Example:
        >>> result = run_calibration_gates(packs, metrics)
        >>> print(generate_gate_report(result))
    """
    lines = []
    lines.append("=" * 80)
    lines.append("CALIBRATION GATE REPORT")
    lines.append("=" * 80)
    lines.append("")

    # Summary
    lines.append(f"Gates Passed: {'✓ YES' if result.passed else '✗ NO'}")
    lines.append(f"Total Violations: {result.summary['total_violations']}")
    lines.append(f"  - Errors: {result.summary['errors']}")
    lines.append(f"  - Warnings: {result.summary['warnings']}")
    lines.append(f"  - Infos: {result.summary['infos']}")
    lines.append("")

    # Errors
    errors = result.get_violations_by_severity(GateSeverity.ERROR)
    if errors:
        lines.append("ERRORS:")
        for v in errors:
            pa_str = f"[{v.policy_area_id}] " if v.policy_area_id else ""
            lines.append(f"  ✗ {pa_str}{v.message}")
            lines.append(f"    Actual: {v.actual_value}, Expected: {v.expected_value}")
            lines.append(f"    Remediation: {v.remediation}")
            lines.append("")

    # Warnings
    warnings = result.get_violations_by_severity(GateSeverity.WARNING)
    if warnings:
        lines.append("WARNINGS:")
        for v in warnings:
            pa_str = f"[{v.policy_area_id}] " if v.policy_area_id else ""
            lines.append(f"  ⚠ {pa_str}{v.message}")
            lines.append(f"    Actual: {v.actual_value}, Expected: {v.expected_value}")
            lines.append(f"    Remediation: {v.remediation}")
            lines.append("")

    lines.append("=" * 80)

    return "\n".join(lines)
```

===== FILE: src/saaaaaa/core/orchestrator/signal_consumption.py =====

```python
"""Signal Consumption Tracking and Verification

This module provides cryptographic proof that signals are actually consumed
during execution, not just loaded into memory.

Key Features:
- Hash chain tracking of pattern matches
- Consumption proof generation for each executor
- Merkle tree verification of pattern origin
- Deterministic proof generation for reproducibility
"""

from __future__ import annotations

import hashlib
import json
import time
from dataclasses import dataclass, field
from typing import TYPE_CHECKING, Any

if TYPE_CHECKING:
    from pathlib import Path
```

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


@dataclass
class SignalConsumptionProof:
    """Cryptographic proof that signals were consumed during execution.

    This class tracks every pattern match and generates a verifiable hash chain
    that proves signal patterns were actually used, not just loaded.

    Attributes:
        executor_id: Unique identifier for the executor
        question_id: Question ID being processed
        policy_area: Policy area of the question
        consumed_patterns: List of (pattern, match_hash) tuples
        proof_chain: Hash chain linking all matches
        timestamp: Unix timestamp of execution
    """

    executor_id: str
    question_id: str
    policy_area: str
    consumed_patterns: list[tuple[str, str]] = field(default_factory=list)
    proof_chain: list[str] = field(default_factory=list)
    timestamp: float = field(default_factory=time.time)

    def record_pattern_match(self, pattern: str, text_segment: str) -> None:
        """Record that a pattern matched text, generating proof.

        Args:
            pattern: The regex pattern that matched
            text_segment: The text segment that matched (truncated to 100 chars)
        """
        # Truncate text segment for proof size
        text_segment = text_segment[:100] if text_segment else ""

        # Generate match hash
        match_hash = hashlib.sha256(
            f"{pattern}|{text_segment}".encode()
        ).hexdigest()

        self.consumed_patterns.append((pattern, match_hash))

        # Update proof chain
        prev_hash = self.proof_chain[-1] if self.proof_chain else "0" * 64
        new_hash = hashlib.sha256(
            f"{prev_hash}|{match_hash}".encode()
        ).hexdigest()
        self.proof_chain.append(new_hash)

        logger.debug(
            "pattern_match_recorded",
            pattern=pattern[:50],
            match_hash=match_hash[:16],
            chain_length=len(self.proof_chain),
        )

    def get_consumption_proof(self) -> dict[str, Any]:
        """Return verifiable proof of signal consumption.

        Returns:
            Dictionary with proof data including:
```

```
            - executor_id, question_id, policy_area
            - patterns_consumed count
            - proof_chain_head (final hash in chain)
            - consumed_hashes (first 10 for verification)
            - timestamp
        """
        return {
            'executor_id': self.executor_id,
            'question_id': self.question_id,
            'policy_area': self.policy_area,
            'patterns_consumed': len(self.consumed_patterns),
            'proof_chain_head': self.proof_chain[-1] if self.proof_chain else None,
            'proof_chain_length': len(self.proof_chain),
            'consumed_hashes': [h for _, h in self.consumed_patterns[:10]],
            'timestamp': self.timestamp,
        }

    def save_to_file(self, output_dir: Path) -> Path:
        """Save consumption proof to JSON file.

        Args:
            output_dir: Directory to save proof files

        Returns:
            Path to the saved proof file
        """
        output_dir.mkdir(parents=True, exist_ok=True)
        proof_file = output_dir / f"{self.question_id}.json"

        with open(proof_file, 'w', encoding='utf-8') as f:
            json.dump(self.get_consumption_proof(), f, indent=2)

        logger.info(
            "consumption_proof_saved",
            question_id=self.question_id,
            proof_file=str(proof_file),
            patterns_consumed=len(self.consumed_patterns),
        )

        return proof_file


def build_merkle_tree(items: list[str]) -> str:
    """Build a simple Merkle tree and return the root hash.

    This is a simplified Merkle tree for verification purposes.
    For production, consider using a full Merkle tree library.

    Args:
        items: List of items to hash

    Returns:
        Hex string of root hash
    """
    if not items:
        return hashlib.sha256(b'').hexdigest()

    # Sort for determinism
    items = sorted(items)

    # Hash each item
    hashes = [
        hashlib.sha256(item.encode('utf-8')).hexdigest()
        for item in items
    ]

    # Build tree bottom-up
    while len(hashes) > 1:
```

```
        if len(hashes) % 2 == 1:
            hashes.append(hashes[-1])  # Duplicate last hash if odd

        next_level = []
        for i in range(0, len(hashes), 2):
            combined = f"{hashes[i]}|{hashes[i+1]}"
            next_hash = hashlib.sha256(combined.encode('utf-8')).hexdigest()
            next_level.append(next_hash)

        hashes = next_level

    return hashes[0]


@dataclass(frozen=True)
class SignalManifest:
    """Cryptographically verifiable signal extraction manifest.

    This manifest provides Merkle roots for all patterns extracted from
    the questionnaire, enabling verification that patterns used during
    execution actually came from the source file.

    Attributes:
        policy_area: Policy area code (e.g., PA01)
        pattern_count: Total number of patterns
        pattern_merkle_root: Merkle root of all patterns
        indicator_merkle_root: Merkle root of indicator patterns
        entity_merkle_root: Merkle root of entity patterns
        extraction_timestamp: Unix timestamp (fixed for determinism)
        source_file_hash: SHA256 of questionnaire_monolith.json
    """

    policy_area: str
    pattern_count: int
    pattern_merkle_root: str
    indicator_merkle_root: str
    entity_merkle_root: str
    extraction_timestamp: float
    source_file_hash: str

    def to_dict(self) -> dict[str, Any]:
        """Convert manifest to dictionary for serialization."""
        return {
            'policy_area': self.policy_area,
            'pattern_count': self.pattern_count,
            'pattern_merkle_root': self.pattern_merkle_root,
            'indicator_merkle_root': self.indicator_merkle_root,
            'entity_merkle_root': self.entity_merkle_root,
            'extraction_timestamp': self.extraction_timestamp,
            'source_file_hash': self.source_file_hash,
        }


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

    Args:
        file_path: Path to file

    Returns:
        Hex string of 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()
```

```python
def generate_signal_manifests(
    questionnaire_data: dict[str, Any],
    source_file_path: Path | None = None,
) -> dict[str, SignalManifest]:
    """Generate signal manifests with Merkle roots for verification.

    Args:
        questionnaire_data: Parsed questionnaire monolith data
        source_file_path: Optional path to source file for hashing

    Returns:
        Dictionary mapping policy area codes to SignalManifest objects
    """
    # Compute source file hash if path provided
    if source_file_path and source_file_path.exists():
        source_hash = compute_file_hash(source_file_path)
    else:
        # Fallback: hash the data itself
        data_str = json.dumps(questionnaire_data, sort_keys=True)
        source_hash = hashlib.sha256(data_str.encode('utf-8')).hexdigest()

    # Fixed timestamp for determinism
    timestamp = 1731258152.0

    manifests = {}
    questions = questionnaire_data.get('blocks', {}).get('micro_questions', [])

    # Group patterns by policy area
    patterns_by_pa: dict[str, dict[str, list[str]]] = {}

    for question in questions:
        pa = question.get('policy_area_id', 'PA01')
        if pa not in patterns_by_pa:
            patterns_by_pa[pa] = {
                'all': [],
                'indicators': [],
                'entities': [],
            }

        for pattern_obj in question.get('patterns', []):
            pattern_str = pattern_obj.get('pattern', '')
            category = pattern_obj.get('category', '')

            if pattern_str:
                patterns_by_pa[pa]['all'].append(pattern_str)

                if category == 'INDICADOR':
                    patterns_by_pa[pa]['indicators'].append(pattern_str)
                elif category == 'FUENTE_OFICIAL':
                    patterns_by_pa[pa]['entities'].append(pattern_str)

    # Build manifests
    for pa, patterns in patterns_by_pa.items():
        manifests[pa] = SignalManifest(
            policy_area=pa,
            pattern_count=len(patterns['all']),
            pattern_merkle_root=build_merkle_tree(patterns['all']),
            indicator_merkle_root=build_merkle_tree(patterns['indicators']),
            entity_merkle_root=build_merkle_tree(patterns['entities']),
            extraction_timestamp=timestamp,
            source_file_hash=source_hash,
        )

        logger.info(
            "signal_manifest_generated",
            policy_area=pa,
            pattern_count=len(patterns['all']),
```

```
            merkle_root=manifests[pa].pattern_merkle_root[:16],
        )

    return manifests
```

===== FILE: src/saaaaaa/core/orchestrator/signal_fallback_fusion.py =====

```
"""Signal Fallback Fusion Module - Intelligent pattern augmentation for PA07-PA10.

This module implements intelligent fallback fusion to address coverage gaps
in PA07-PA10 by selectively borrowing patterns from high-coverage policy areas.

Key Features:
- Semantic similarity-based pattern selection
- Cross-PA pattern sharing with provenance tracking
- Fusion quality gates (prevents over-fusion)
- Dynamic threshold adjustment for low-coverage PAs
- Audit trail for fused patterns

SOTA Requirements:
- Solves PA07-PA10 coverage gap without degrading precision
- Maintains fingerprint integrity via soft-alias pattern
- Supports quality metrics monitoring
"""

from __future__ import annotations

from dataclasses import dataclass, field
from typing import TYPE_CHECKING, Any

if TYPE_CHECKING:
    from .signals import SignalPack
    from .signal_quality_metrics import SignalQualityMetrics

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


@dataclass
class FusionStrategy:
    """Fusion strategy configuration.

    Attributes:
        min_source_patterns: Minimum patterns required in source PA for fusion
        max_fusion_ratio: Maximum fusion patterns / original patterns ratio
        similarity_threshold: Minimum semantic similarity for pattern selection
        preserve_thresholds: Whether to preserve original confidence thresholds
        fusion_provenance: Whether to track pattern provenance
    """
    min_source_patterns: int = 20
    max_fusion_ratio: float = 0.50  # Max 50% augmentation
    similarity_threshold: float = 0.30  # Relaxed for cross-domain
    preserve_thresholds: bool = True
    fusion_provenance: bool = True


@dataclass
class FusedPattern:
    """Container for fused pattern with provenance.

    Attributes:
        pattern: Pattern string
        source_pa: Source policy area ID
        target_pa: Target policy area ID
        similarity_score: Semantic similarity score (0.0-1.0)
```

```python
        fusion_method: Fusion method used
        metadata: Additional metadata
    """
    pattern: str
    source_pa: str
    target_pa: str
    similarity_score: float
    fusion_method: str
    metadata: dict[str, Any] = field(default_factory=dict)


def compute_pattern_similarity(
    pattern1: str,
    pattern2: str,
) -> float:
    """
    Compute semantic similarity between two patterns.

    This is a simplified similarity metric based on:
    - Token overlap (Jaccard similarity)
    - Common n-grams
    - Length similarity

    Args:
        pattern1: First pattern string
        pattern2: Second pattern string

    Returns:
        Similarity score (0.0-1.0)

    Example:
        >>> sim = compute_pattern_similarity("tierras", "territorio")
        >>> print(f"Similarity: {sim:.2f}")
    """
    # Normalize patterns
    p1_tokens = set(pattern1.lower().split())
    p2_tokens = set(pattern2.lower().split())

    if not p1_tokens or not p2_tokens:
        return 0.0

    # Jaccard similarity
    intersection = p1_tokens & p2_tokens
    union = p1_tokens | p2_tokens
    jaccard = len(intersection) / len(union) if union else 0.0

    # Character n-gram similarity (trigrams)
    def get_trigrams(text: str) -> set[str]:
        return {text[i:i+3] for i in range(len(text)-2)}

    trigrams1 = get_trigrams(pattern1.lower())
    trigrams2 = get_trigrams(pattern2.lower())

    if trigrams1 and trigrams2:
        trigram_intersection = trigrams1 & trigrams2
        trigram_union = trigrams1 | trigrams2
        trigram_sim = len(trigram_intersection) / len(trigram_union)
    else:
        trigram_sim = 0.0

    # Length similarity
    len_sim = 1.0 - abs(len(pattern1) - len(pattern2)) / max(len(pattern1), len(pattern2))

    # Weighted average
    similarity = 0.5 * jaccard + 0.3 * trigram_sim + 0.2 * len_sim

    return similarity
```

```python
def select_fusion_candidates(
    source_patterns: list[str],
    target_patterns: list[str],
    strategy: FusionStrategy,
) -> list[str]:
    """
    Select fusion candidate patterns from source PA.

    This implements intelligent pattern selection:
    1. Filter out patterns already in target
    2. Compute similarity to target patterns
    3. Select high-similarity candidates
    4. Limit by max_fusion_ratio

    Args:
        source_patterns: Patterns from high-coverage PA
        target_patterns: Patterns from low-coverage PA
        strategy: Fusion strategy configuration

    Returns:
        List of selected fusion candidate patterns

    Example:
        >>> source = ["tierras", "territorio", "reforma agraria"]
        >>> target = ["tierras"]
        >>> strategy = FusionStrategy()
        >>> candidates = select_fusion_candidates(source, target, strategy)
        >>> print(candidates)
    """
    # Filter out patterns already in target
    target_set = set(p.lower() for p in target_patterns)
    novel_patterns = [
        p for p in source_patterns
        if p.lower() not in target_set
    ]

    # Compute max similarity to any target pattern
    pattern_similarities = []
    for novel_pattern in novel_patterns:
        max_sim = 0.0
        for target_pattern in target_patterns:
            sim = compute_pattern_similarity(novel_pattern, target_pattern)
            max_sim = max(max_sim, sim)

        if max_sim >= strategy.similarity_threshold:
            pattern_similarities.append((novel_pattern, max_sim))

    # Sort by similarity (descending)
    pattern_similarities.sort(key=lambda x: x[1], reverse=True)

    # Limit by max_fusion_ratio
    max_fusion_count = int(len(target_patterns) * strategy.max_fusion_ratio)
    selected_patterns = [
        pattern for pattern, sim in pattern_similarities[:max_fusion_count]
    ]

    logger.debug(
        "fusion_candidates_selected",
        source_count=len(source_patterns),
        target_count=len(target_patterns),
        novel_count=len(novel_patterns),
        selected_count=len(selected_patterns),
    )

    return selected_patterns
```

```
def fuse_signal_packs(
    target_pack: SignalPack,
    source_packs: list[SignalPack],
    target_pa_id: str,
    strategy: FusionStrategy | None = None,
) -> tuple[SignalPack, list[FusedPattern]]:
    """
    Fuse patterns from source packs into target pack.

    This implements the intelligent fallback fusion algorithm:
    1. Filter source packs by min_source_patterns
    2. Select fusion candidates from each source
    3. Augment target pack with fused patterns
    4. Preserve original patterns and metadata
    5. Track fusion provenance

    Args:
        target_pack: Low-coverage SignalPack to augment
        source_packs: List of high-coverage SignalPacks to borrow from
        target_pa_id: Target policy area ID (PA07-PA10)
        strategy: Fusion strategy (uses default if None)

    Returns:
        Tuple of (fused_pack, fusion_provenance)

    Example:
        >>> pa07_pack = build_signal_pack_from_monolith("PA07")
        >>> pa01_pack = build_signal_pack_from_monolith("PA01")
        >>> fused_pack, provenance = fuse_signal_packs(pa07_pack, [pa01_pack], "PA07")
        >>> print(f"Original: {len(pa07_pack.patterns)}, Fused:
{len(fused_pack.patterns)}")
    """
    if strategy is None:
        strategy = FusionStrategy()

    # Filter source packs by min_source_patterns
    eligible_sources = [
        pack for pack in source_packs
        if len(pack.patterns) >= strategy.min_source_patterns
    ]

    if not eligible_sources:
        logger.warning(
            "no_eligible_fusion_sources",
            target_pa=target_pa_id,
            min_patterns=strategy.min_source_patterns,
        )
        return target_pack, []

    # Collect fusion candidates from all sources
    all_fusion_patterns: list[FusedPattern] = []
    fused_pattern_strings = set()

    for source_pack in eligible_sources:
        source_pa_id = source_pack.metadata.get("original_policy_area", "unknown")

        # Select fusion candidates
        candidates = select_fusion_candidates(
            source_pack.patterns,
            target_pack.patterns,
            strategy,
        )

        # Create FusedPattern objects
        for pattern in candidates:
            if pattern in fused_pattern_strings:
                continue  # Skip duplicates across sources
```

```python
        # Compute similarity to target patterns
        max_sim = 0.0
        for target_pattern in target_pack.patterns:
            sim = compute_pattern_similarity(pattern, target_pattern)
            max_sim = max(max_sim, sim)

        fused_pattern = FusedPattern(
            pattern=pattern,
            source_pa=source_pa_id,
            target_pa=target_pa_id,
            similarity_score=max_sim,
            fusion_method="intelligent_fallback",
            metadata={
                "source_fingerprint": source_pack.source_fingerprint,
                "fusion_timestamp": "2025-01-18T00:00:00Z",
            },
        )

        all_fusion_patterns.append(fused_pattern)
        fused_pattern_strings.add(pattern)

# Augment target pack
original_pattern_count = len(target_pack.patterns)
augmented_patterns = target_pack.patterns + list(fused_pattern_strings)

# Similarly augment indicators and entities (proportionally)
augmented_indicators = target_pack.indicators.copy()
augmented_entities = target_pack.entities.copy()

# Update metadata to reflect fusion
fusion_metadata = {
    "fusion_enabled": True,
    "original_pattern_count": original_pattern_count,
    "fused_pattern_count": len(fused_pattern_strings),
    "fusion_ratio": len(fused_pattern_strings) / max(original_pattern_count, 1),
    "fusion_sources": [
        pack.metadata.get("original_policy_area", "unknown")
        for pack in eligible_sources
    ],
    "fusion_strategy": {
        "min_source_patterns": strategy.min_source_patterns,
        "max_fusion_ratio": strategy.max_fusion_ratio,
        "similarity_threshold": strategy.similarity_threshold,
    },
}

# Create fused pack (preserving original structure)
fused_pack = SignalPack(
    version=target_pack.version,
    policy_area=target_pack.policy_area,
    patterns=augmented_patterns[:200],  # Limit for performance
    indicators=augmented_indicators[:50],
    regex=target_pack.regex.copy(),
    entities=augmented_entities[:100],
    thresholds=target_pack.thresholds.copy(),
    ttl_s=target_pack.ttl_s,
    source_fingerprint=target_pack.source_fingerprint,
    metadata={
        **target_pack.metadata,
        "fusion": fusion_metadata,
    },
)

logger.info(
    "signal_packs_fused",
    target_pa=target_pa_id,
    original_patterns=original_pattern_count,
    fused_patterns=len(fused_pattern_strings),
```

```
        fusion_ratio=round(fusion_metadata["fusion_ratio"], 3),
        sources=len(eligible_sources),
    )

    return fused_pack, all_fusion_patterns


def apply_intelligent_fallback_fusion(
    signal_packs: dict[str, SignalPack],
    metrics_by_pa: dict[str, SignalQualityMetrics],
    strategy: FusionStrategy | None = None,
) -> dict[str, SignalPack]:
    """
    Apply intelligent fallback fusion to low-coverage policy areas.

    This is the main entry point for PA07-PA10 coverage gap resolution.

    Args:
        signal_packs: Dict mapping policy_area_id to SignalPack
        metrics_by_pa: Dict mapping policy_area_id to SignalQualityMetrics
        strategy: Fusion strategy (uses default if None)

    Returns:
        Updated signal_packs with fusion applied to low-coverage PAs

    Example:
        >>> packs = build_all_signal_packs()
        >>> metrics = {pa: compute_signal_quality_metrics(pack, pa) for pa, pack in
packs.items()}
        >>> fused_packs = apply_intelligent_fallback_fusion(packs, metrics)
        >>> print(f"Fusion applied to {len(fused_packs)} PAs")
    """
    if strategy is None:
        strategy = FusionStrategy()

    # Identify low-coverage PAs (typically PA07-PA10)
    low_coverage_pas = [
        pa for pa, metrics in metrics_by_pa.items()
        if metrics.coverage_tier in ("SPARSE", "ADEQUATE")
    ]

    # Identify high-coverage PAs (typically PA01-PA06)
    high_coverage_pas = [
        pa for pa, metrics in metrics_by_pa.items()
        if metrics.coverage_tier in ("GOOD", "EXCELLENT")
    ]

    if not low_coverage_pas or not high_coverage_pas:
        logger.info(
            "fusion_skipped_no_candidates",
            low_coverage_count=len(low_coverage_pas),
            high_coverage_count=len(high_coverage_pas),
        )
        return signal_packs

    # Prepare source packs
    source_packs = [signal_packs[pa] for pa in high_coverage_pas]

    # Apply fusion to each low-coverage PA
    fused_packs = signal_packs.copy()
    total_fused_patterns = 0

    for target_pa in low_coverage_pas:
        target_pack = signal_packs[target_pa]

        # Apply fusion
        fused_pack, provenance = fuse_signal_packs(
            target_pack,
```

```
            source_packs,
            target_pa,
            strategy,
        )

        fused_packs[target_pa] = fused_pack
        total_fused_patterns += len(provenance)

    logger.info(
        "intelligent_fallback_fusion_applied",
        low_coverage_pas=low_coverage_pas,
        high_coverage_pas=high_coverage_pas,
        total_fused_patterns=total_fused_patterns,
    )

    return fused_packs


def generate_fusion_audit_report(
    signal_packs: dict[str, SignalPack]
) -> dict[str, Any]:
    """
    Generate audit report for fusion operations.

    Args:
        signal_packs: Dict mapping policy_area_id to SignalPack

    Returns:
        Fusion audit report with provenance and quality metrics

    Example:
        >>> fused_packs = apply_intelligent_fallback_fusion(packs, metrics)
        >>> audit_report = generate_fusion_audit_report(fused_packs)
        >>> print(json.dumps(audit_report, indent=2))
    """
    fusion_enabled_pas = []
    fusion_summary = {}

    for pa, pack in signal_packs.items():
        fusion_metadata = pack.metadata.get("fusion", {})
        if fusion_metadata.get("fusion_enabled"):
            fusion_enabled_pas.append(pa)
            fusion_summary[pa] = {
                "original_patterns": fusion_metadata["original_pattern_count"],
                "fused_patterns": fusion_metadata["fused_pattern_count"],
                "fusion_ratio": round(fusion_metadata["fusion_ratio"], 3),
                "fusion_sources": fusion_metadata["fusion_sources"],
            }

    report = {
        "fusion_enabled_pas": fusion_enabled_pas,
        "fusion_summary": fusion_summary,
        "total_fused_patterns": sum(
            s["fused_patterns"] for s in fusion_summary.values()
        ),
        "avg_fusion_ratio": (
            sum(s["fusion_ratio"] for s in fusion_summary.values()) / len(fusion_summary)
            if fusion_summary else 0.0
        ),
    }

    logger.info(
        "fusion_audit_report_generated",
        fusion_enabled_pas=len(fusion_enabled_pas),
        total_fused_patterns=report["total_fused_patterns"],
    )

    return report
```

```
===== FILE: src/saaaaaa/core/orchestrator/signal_loader.py =====
"""Signal Loader Module - Extract patterns from questionnaire_monolith.json

This module implements Phase 1 of the Signal Integration Plan by extracting
REAL patterns from the questionnaire_monolith.json file and building SignalPack
objects for each of the 10 policy areas.

Key Features:
- Extracts ~2200 patterns from 300 micro_questions
- Groups patterns by policy_area_id (PA01-PA10)
- Categorizes patterns by type (TEMPORAL, INDICADOR, FUENTE_OFICIAL, etc.)
- Builds versioned SignalPack objects with fingerprints
- Computes source fingerprints using blake3/hashlib
"""

from __future__ import annotations

import hashlib
import json
from typing import TYPE_CHECKING, Any

if TYPE_CHECKING:
    from .questionnaire import CanonicalQuestionnaire

try:
    import blake3
    BLAKE3_AVAILABLE = True
except ImportError:
    BLAKE3_AVAILABLE = False

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

from .signal_consumption import SignalManifest, generate_signal_manifests
from .signals import SignalPack


def compute_fingerprint(content: str | bytes) -> str:
    """
    Compute fingerprint of content using blake3 or sha256 fallback.

    Args:
        content: String or bytes to hash

    Returns:
        Hex string of hash
    """
    if isinstance(content, str):
        content = content.encode('utf-8')

    if BLAKE3_AVAILABLE:
        return blake3.blake3(content).hexdigest()
    else:
        return hashlib.sha256(content).hexdigest()


# DEPRECATED: Re-exported from factory.py for backward compatibility
# Do NOT create additional implementations - this is the single source


def extract_patterns_by_policy_area(
    monolith: dict[str, Any]
) -> dict[str, list[dict[str, Any]]]:
```

```
    """
    Extract patterns grouped by policy area.

    Args:
        monolith: Loaded questionnaire monolith data

    Returns:
        Dict mapping policy_area_id to list of patterns
    """
    questions = monolith.get('blocks', {}).get('micro_questions', [])

    patterns_by_pa = {}
    for question in questions:
        policy_area = question.get('policy_area_id', 'PA01')
        patterns = question.get('patterns', [])

        if policy_area not in patterns_by_pa:
            patterns_by_pa[policy_area] = []

        patterns_by_pa[policy_area].extend(patterns)

    logger.info(
        "patterns_extracted_by_policy_area",
        policy_areas=len(patterns_by_pa),
        total_patterns=sum(len(p) for p in patterns_by_pa.values()),
    )

    return patterns_by_pa


def categorize_patterns(
    patterns: list[dict[str, Any]]
) -> dict[str, list[str]]:
    """
    Categorize patterns by their category field.

    Args:
        patterns: List of pattern objects

    Returns:
        Dict with categorized pattern strings:
        - all_patterns: All non-TEMPORAL patterns
        - indicators: INDICADOR patterns
        - sources: FUENTE_OFICIAL patterns
        - temporal: TEMPORAL patterns
    """
    categorized = {
        'all_patterns': [],
        'indicators': [],
        'sources': [],
        'temporal': [],
        'entities': [],
    }

    for pattern_obj in patterns:
        pattern_str = pattern_obj.get('pattern', '')
        category = pattern_obj.get('category', '')

        if not pattern_str:
            continue

        # All non-temporal patterns
        if category != 'TEMPORAL':
            categorized['all_patterns'].append(pattern_str)

        # Category-specific
        if category == 'INDICADOR':
            categorized['indicators'].append(pattern_str)
```

```
        elif category == 'FUENTE_OFICIAL':
            categorized['sources'].append(pattern_str)
            # Sources are also entities
            # Extract entity names from pattern (simplified)
            parts = pattern_str.split('|')
            categorized['entities'].extend(p.strip() for p in parts if p.strip())
        elif category == 'TEMPORAL':
            categorized['temporal'].append(pattern_str)

    # Deduplicate
    for key in categorized:
        categorized[key] = list(set(categorized[key]))

    return categorized


def extract_thresholds(patterns: list[dict[str, Any]]) -> dict[str, float]:
    """
    Extract threshold values from pattern confidence_weight fields.

    Args:
        patterns: List of pattern objects

    Returns:
        Dict with threshold values
    """
    confidence_weights = [
        p.get('confidence_weight', 0.85)
        for p in patterns
        if 'confidence_weight' in p
    ]

    if confidence_weights:
        min_confidence = min(confidence_weights)
        max_confidence = max(confidence_weights)
        avg_confidence = sum(confidence_weights) / len(confidence_weights)
    else:
        min_confidence = 0.85
        max_confidence = 0.85
        avg_confidence = 0.85

    return {
        'min_confidence': round(min_confidence, 2),
        'max_confidence': round(max_confidence, 2),
        'avg_confidence': round(avg_confidence, 2),
        'min_evidence': 0.70,  # Derived from scoring requirements
    }


def get_git_sha() -> str:
    """
    Get current git commit SHA (short form).

    Returns:
        Short SHA or 'unknown' if not in git repo
    """
    try:
        import subprocess
        result = subprocess.run(
            ['git', 'rev-parse', '--short', 'HEAD'],
            check=False, capture_output=True,
            text=True,
            timeout=2,
        )
        if result.returncode == 0:
            return result.stdout.strip()
    except Exception:
        pass
```

```
    return 'unknown'


def build_signal_pack_from_monolith(
    policy_area: str,
    monolith: dict[str, Any] | None = None,
    *,
    questionnaire: CanonicalQuestionnaire | None = None,
) -> SignalPack:
    """
    Build SignalPack for a specific policy area from questionnaire monolith.

    This extracts REAL patterns from the questionnaire_monolith.json file and
    constructs a versioned SignalPack with proper categorization.

    Args:
        policy_area: Policy area code (PA01-PA10)
        monolith: DEPRECATED - Optional pre-loaded monolith data (use questionnaire
parameter instead)
        questionnaire: Optional CanonicalQuestionnaire instance (recommended, loads from
canonical if None)

    Returns:
        SignalPack object with extracted patterns

    Example:
        >>> from saaaaaa.core.orchestrator.questionnaire import load_questionnaire
        >>> canonical = load_questionnaire()
        >>> pack = build_signal_pack_from_monolith("PA01", questionnaire=canonical)
        >>> print(f"Patterns: {len(pack.patterns)}")
        >>> print(f"Indicators: {len(pack.indicators)}")
    """
    # Import here to avoid circular dependency
    from .questionnaire import load_questionnaire

    # Handle legacy monolith parameter
    if monolith is not None:
        import warnings
        warnings.warn(
            "build_signal_pack_from_monolith: 'monolith' parameter is DEPRECATED. "
            "Use 'questionnaire' parameter with CanonicalQuestionnaire instead.",
            DeprecationWarning,
            stacklevel=2
        )
        # Use legacy monolith if provided
        monolith_data = monolith
    elif questionnaire is not None:
        # Use canonical questionnaire (preferred)
        monolith_data = dict(questionnaire.data)
    else:
        # Load from canonical loader
        canonical = load_questionnaire()
        monolith_data = dict(canonical.data)

    # Extract patterns by policy area
    patterns_by_pa = extract_patterns_by_policy_area(monolith_data)

    if policy_area not in patterns_by_pa:
        logger.warning(
            "policy_area_not_found",
            policy_area=policy_area,
            available=list(patterns_by_pa.keys()),
        )
        # Return empty signal pack
        return SignalPack(
            version="1.0.0",
            policy_area="fiscal",  # Default PolicyArea type
```

```python
            patterns=[],
            indicators=[],
            regex=[],
            entities=[],
            thresholds={},
        )

    # Get patterns for this policy area
    raw_patterns = patterns_by_pa[policy_area]

    # Categorize patterns
    categorized = categorize_patterns(raw_patterns)

    # Extract thresholds
    thresholds = extract_thresholds(raw_patterns)

    # Compute source fingerprint
    monolith_str = json.dumps(monolith_data, sort_keys=True)
    source_fingerprint = compute_fingerprint(monolith_str)

    # Build version string (must be semantic X.Y.Z format)
    git_sha = get_git_sha()
    # Use 1.0.0 as base version (git sha stored in metadata)
    version = "1.0.0"

    # Regex patterns are all patterns (for now)
    regex_patterns = categorized['all_patterns'][:100]  # Limit for performance

    # Map policy area to PolicyArea type (using fiscal as default)
    # The SignalPack PolicyArea type is limited, so we use fiscal as a placeholder
    policy_area_type = "fiscal"

    # Build SignalPack
    signal_pack = SignalPack(
        version=version,
        policy_area=policy_area_type,
        patterns=categorized['all_patterns'][:200],  # Limit for performance
        indicators=categorized['indicators'][:50],
        regex=regex_patterns,
        entities=categorized['entities'][:100],
        thresholds=thresholds,
        ttl_s=86400,  # 24 hours
        source_fingerprint=source_fingerprint[:32],  # Truncate for readability
        metadata={
            'original_policy_area': policy_area,
            'total_raw_patterns': len(raw_patterns),
            'categorized_counts': {
                key: len(val) for key, val in categorized.items()
            },
            'git_sha': git_sha,
        }
    )

    logger.info(
        "signal_pack_built",
        policy_area=policy_area,
        version=version,
        patterns=len(signal_pack.patterns),
        indicators=len(signal_pack.indicators),
        entities=len(signal_pack.entities),
    )

    return signal_pack


def build_all_signal_packs(
    monolith: dict[str, Any] | None = None,
    *,
```

```
    questionnaire: CanonicalQuestionnaire | None = None,
) -> dict[str, SignalPack]:
    """
    Build SignalPacks for all policy areas.

    Args:
        monolith: DEPRECATED - Optional pre-loaded monolith data (use questionnaire
parameter instead)
        questionnaire: Optional CanonicalQuestionnaire instance (recommended, loads from
canonical if None)

    Returns:
        Dict mapping policy_area_id to SignalPack

    Example:
        >>> from saaaaaa.core.orchestrator.questionnaire import load_questionnaire
        >>> canonical = load_questionnaire()
        >>> packs = build_all_signal_packs(questionnaire=canonical)
        >>> print(f"Built {len(packs)} signal packs")
    """
    # Import here to avoid circular dependency
    from .questionnaire import load_questionnaire

    # Handle legacy monolith parameter and ensure questionnaire is loaded only once
    if monolith is not None:
        import warnings
        warnings.warn(
            "build_all_signal_packs: 'monolith' parameter is DEPRECATED. "
            "Use 'questionnaire' parameter with CanonicalQuestionnaire instead.",
            DeprecationWarning,
            stacklevel=2
        )
    elif questionnaire is None:
        # Load questionnaire once to avoid redundant I/O in loop
        questionnaire = load_questionnaire()

    policy_areas = [f"PA{i:02d}" for i in range(1, 11)]

    signal_packs = {}
    for pa in policy_areas:
        signal_packs[pa] = build_signal_pack_from_monolith(
            pa, monolith=monolith, questionnaire=questionnaire
        )

    logger.info(
        "all_signal_packs_built",
        count=len(signal_packs),
        policy_areas=list(signal_packs.keys()),
    )

    return signal_packs


def build_signal_manifests(
    monolith: dict[str, Any] | None = None,
    *,
    questionnaire: CanonicalQuestionnaire | None = None,
) -> dict[str, SignalManifest]:
    """
    Build signal manifests with Merkle roots for verification.

    Args:
        monolith: DEPRECATED - Optional pre-loaded monolith data (use questionnaire
parameter instead)
        questionnaire: Optional CanonicalQuestionnaire instance (recommended, loads from
canonical if None)

    Returns:
```

```
        Dict mapping policy_area_id to SignalManifest

    Example:
        >>> from saaaaaa.core.orchestrator.questionnaire import load_questionnaire
        >>> canonical = load_questionnaire()
        >>> manifests = build_signal_manifests(questionnaire=canonical)
        >>> print(f"Built {len(manifests)} manifests")
    """
    # Import here to avoid circular dependency
    from .questionnaire import QUESTIONNAIRE_PATH, load_questionnaire

    # Handle legacy monolith parameter
    if monolith is not None:
        import warnings
        warnings.warn(
            "build_signal_manifests: 'monolith' parameter is DEPRECATED. "
            "Use 'questionnaire' parameter with CanonicalQuestionnaire instead.",
            DeprecationWarning,
            stacklevel=2
        )
        monolith_data = monolith
    elif questionnaire is not None:
        # Use canonical questionnaire (preferred)
        monolith_data = dict(questionnaire.data)
    else:
        # Load from canonical loader
        canonical = load_questionnaire()
        monolith_data = dict(canonical.data)

    # Always use canonical path
    monolith_path = QUESTIONNAIRE_PATH
    manifests = generate_signal_manifests(monolith_data, monolith_path)

    logger.info(
        "signal_manifests_built",
        count=len(manifests),
        policy_areas=list(manifests.keys()),
    )

    return manifests

===== FILE: src/saaaaaa/core/orchestrator/signal_quality_metrics.py =====
"""Signal Quality Metrics Module - Observability for PA coverage analysis.

This module implements quality metrics monitoring for policy area coverage,
specifically designed to detect and measure PA07-PA10 coverage gaps.

Key Features:
- Pattern density metrics (patterns per policy area)
- Threshold calibration tracking (min_confidence, min_evidence)
- Entity coverage analysis (institutional completeness)
- Temporal freshness monitoring (TTL, valid_from/valid_to)
- Coverage gap detection (PA07-PA10 vs PA01-PA06 comparison)

SOTA Requirements:
- Observability for PA coverage gaps
- Quality gates for calibration drift
- Metrics for intelligent fallback fusion
"""

from __future__ import annotations

from dataclasses import dataclass, field
from typing import TYPE_CHECKING, Any

if TYPE_CHECKING:
    from .signals import SignalPack
```

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


@dataclass
class SignalQualityMetrics:
    """Quality metrics for a single SignalPack.

    Attributes:
        policy_area_id: Policy area identifier (PA01-PA10)
        pattern_count: Total number of patterns
        indicator_count: Total number of indicators
        entity_count: Total number of entities
        regex_count: Total number of regex patterns
        threshold_min_confidence: Minimum confidence threshold
        threshold_min_evidence: Minimum evidence threshold
        ttl_hours: Time-to-live in hours
        has_temporal_bounds: Whether valid_from/valid_to are set
        pattern_density: Patterns per 100 tokens (estimated)
        entity_coverage_ratio: Entities / patterns ratio
        fingerprint: Source fingerprint
        metadata: Additional metadata
    """
    policy_area_id: str
    pattern_count: int
    indicator_count: int
    entity_count: int
    regex_count: int
    threshold_min_confidence: float
    threshold_min_evidence: float
    ttl_hours: float
    has_temporal_bounds: bool
    pattern_density: float
    entity_coverage_ratio: float
    fingerprint: str
    metadata: dict[str, Any] = field(default_factory=dict)

    @property
    def is_high_quality(self) -> bool:
        """Check if signal pack meets high-quality thresholds.

        High-quality criteria:
        - At least 15 patterns
        - At least 3 indicators
        - At least 3 entities
        - Min confidence >= 0.75
        - Min evidence >= 0.70
        - Entity coverage ratio >= 0.15
        """
        return (
            self.pattern_count >= 15
            and self.indicator_count >= 3
            and self.entity_count >= 3
            and self.threshold_min_confidence >= 0.75
            and self.threshold_min_evidence >= 0.70
            and self.entity_coverage_ratio >= 0.15
        )

    @property
    def coverage_tier(self) -> str:
        """Classify coverage tier based on pattern count.

        Tiers:
        - EXCELLENT: >= 30 patterns
```

```python
        - GOOD: >= 20 patterns
        - ADEQUATE: >= 15 patterns
        - SPARSE: < 15 patterns
        """
        if self.pattern_count >= 30:
            return "EXCELLENT"
        elif self.pattern_count >= 20:
            return "GOOD"
        elif self.pattern_count >= 15:
            return "ADEQUATE"
        else:
            return "SPARSE"


@dataclass
class CoverageGapAnalysis:
    """Coverage gap analysis comparing PA groups.

    Attributes:
        high_coverage_pas: List of PA IDs with high coverage (typically PA01-PA06)
        low_coverage_pas: List of PA IDs with low coverage (typically PA07-PA10)
        coverage_delta: Average pattern count difference
        threshold_delta: Average confidence threshold difference
        gap_severity: Classification of gap severity
        recommendations: List of recommended actions
    """
    high_coverage_pas: list[str]
    low_coverage_pas: list[str]
    coverage_delta: float
    threshold_delta: float
    gap_severity: str
    recommendations: list[str] = field(default_factory=list)

    @property
    def requires_fallback_fusion(self) -> bool:
        """Check if coverage gap requires intelligent fallback fusion."""
        return self.gap_severity in ("CRITICAL", "SEVERE")


def compute_signal_quality_metrics(
    signal_pack: SignalPack,
    policy_area_id: str,
) -> SignalQualityMetrics:
    """
    Compute quality metrics for a SignalPack.

    Args:
        signal_pack: SignalPack object to analyze
        policy_area_id: Policy area identifier (PA01-PA10)

    Returns:
        SignalQualityMetrics object

    Example:
        >>> pack = build_signal_pack_from_monolith("PA07")
        >>> metrics = compute_signal_quality_metrics(pack, "PA07")
        >>> print(f"Coverage tier: {metrics.coverage_tier}")
        >>> print(f"High quality: {metrics.is_high_quality}")
    """
    pattern_count = len(signal_pack.patterns)
    indicator_count = len(signal_pack.indicators)
    entity_count = len(signal_pack.entities)
    regex_count = len(signal_pack.regex)

    # Extract thresholds
    threshold_min_confidence = signal_pack.thresholds.get("min_confidence", 0.85)
    threshold_min_evidence = signal_pack.thresholds.get("min_evidence", 0.70)
```

```
    # Convert TTL to hours
    ttl_hours = signal_pack.ttl_s / 3600.0 if signal_pack.ttl_s else 24.0

    # Check temporal bounds
    has_temporal_bounds = bool(
        signal_pack.metadata.get("valid_from") or
        hasattr(signal_pack, 'valid_from') and signal_pack.valid_from  # type: ignore
    )

    # Estimate pattern density (patterns per 100 tokens)
    # Assuming average pattern length of 3 tokens
    estimated_tokens = pattern_count * 3
    pattern_density = (pattern_count / max(estimated_tokens, 1)) * 100

    # Entity coverage ratio
    entity_coverage_ratio = entity_count / max(pattern_count, 1)

    metrics = SignalQualityMetrics(
        policy_area_id=policy_area_id,
        pattern_count=pattern_count,
        indicator_count=indicator_count,
        entity_count=entity_count,
        regex_count=regex_count,
        threshold_min_confidence=threshold_min_confidence,
        threshold_min_evidence=threshold_min_evidence,
        ttl_hours=ttl_hours,
        has_temporal_bounds=has_temporal_bounds,
        pattern_density=pattern_density,
        entity_coverage_ratio=entity_coverage_ratio,
        fingerprint=signal_pack.source_fingerprint,
        metadata={
            "version": signal_pack.version,
            "original_metadata": signal_pack.metadata,
        },
    )

    logger.debug(
        "signal_quality_metrics_computed",
        policy_area_id=policy_area_id,
        coverage_tier=metrics.coverage_tier,
        is_high_quality=metrics.is_high_quality,
        pattern_count=pattern_count,
    )

    return metrics


def analyze_coverage_gaps(
    metrics_by_pa: dict[str, SignalQualityMetrics]
) -> CoverageGapAnalysis:
    """
    Analyze coverage gaps between PA groups (PA01-PA06 vs PA07-PA10).

    This implements the coverage gap detection algorithm for SOTA requirements.

    Args:
        metrics_by_pa: Dict mapping policy_area_id to SignalQualityMetrics

    Returns:
        CoverageGapAnalysis object

    Example:
        >>> packs = build_all_signal_packs()
        >>> metrics = {pa: compute_signal_quality_metrics(pack, pa) for pa, pack in
packs.items()}
        >>> gap_analysis = analyze_coverage_gaps(metrics)
        >>> print(f"Gap severity: {gap_analysis.gap_severity}")
        >>> print(f"Requires fallback: {gap_analysis.requires_fallback_fusion}")
```

```
    """
    # Split into high-coverage and low-coverage groups
    pa01_pa06 = [f"PA{i:02d}" for i in range(1, 7)]
    pa07_pa10 = [f"PA{i:02d}" for i in range(7, 11)]

    high_coverage_metrics = [
        metrics_by_pa[pa] for pa in pa01_pa06 if pa in metrics_by_pa
    ]
    low_coverage_metrics = [
        metrics_by_pa[pa] for pa in pa07_pa10 if pa in metrics_by_pa
    ]

    if not high_coverage_metrics or not low_coverage_metrics:
        return CoverageGapAnalysis(
            high_coverage_pas=[],
            low_coverage_pas=[],
            coverage_delta=0.0,
            threshold_delta=0.0,
            gap_severity="UNKNOWN",
            recommendations=["Insufficient data for gap analysis"],
        )

    # Compute average pattern counts
    high_avg_patterns = sum(m.pattern_count for m in high_coverage_metrics) /
len(high_coverage_metrics)
    low_avg_patterns = sum(m.pattern_count for m in low_coverage_metrics) /
len(low_coverage_metrics)
    coverage_delta = high_avg_patterns - low_avg_patterns

    # Compute average confidence thresholds
    high_avg_confidence = sum(m.threshold_min_confidence for m in high_coverage_metrics) /
 len(high_coverage_metrics)
    low_avg_confidence = sum(m.threshold_min_confidence for m in low_coverage_metrics) /
len(low_coverage_metrics)
    threshold_delta = high_avg_confidence - low_avg_confidence

    # Classify gap severity
    if coverage_delta >= 50:
        gap_severity = "CRITICAL"
    elif coverage_delta >= 30:
        gap_severity = "SEVERE"
    elif coverage_delta >= 15:
        gap_severity = "MODERATE"
    elif coverage_delta >= 5:
        gap_severity = "MINOR"
    else:
        gap_severity = "NEGLIGIBLE"

    # Generate recommendations
    recommendations = []
    if gap_severity in ("CRITICAL", "SEVERE"):
        recommendations.append("Enable intelligent fallback fusion for PA07-PA10")
        recommendations.append("Review pattern extraction for low-coverage PAs")
        recommendations.append("Consider cross-PA pattern sharing for common terms")

    if threshold_delta > 0.05:
        recommendations.append("Recalibrate confidence thresholds for consistency")

    # Identify specific low-coverage PAs
    sparse_pas = [
        m.policy_area_id for m in low_coverage_metrics
        if m.coverage_tier == "SPARSE"
    ]
    if sparse_pas:
        recommendations.append(f"Boost pattern extraction for: {', '.join(sparse_pas)}")

    analysis = CoverageGapAnalysis(
        high_coverage_pas=[m.policy_area_id for m in high_coverage_metrics],
```

```
        low_coverage_pas=[m.policy_area_id for m in low_coverage_metrics],
        coverage_delta=coverage_delta,
        threshold_delta=threshold_delta,
        gap_severity=gap_severity,
        recommendations=recommendations,
    )

    logger.info(
        "coverage_gap_analysis_completed",
        gap_severity=gap_severity,
        coverage_delta=coverage_delta,
        requires_fallback=analysis.requires_fallback_fusion,
    )

    return analysis


def generate_quality_report(
    metrics_by_pa: dict[str, SignalQualityMetrics]
) -> dict[str, Any]:
    """
    Generate comprehensive quality report for all policy areas.

    Args:
        metrics_by_pa: Dict mapping policy_area_id to SignalQualityMetrics

    Returns:
        Quality report dict with:
        - summary: Overall statistics
        - by_policy_area: Per-PA metrics
        - coverage_gap_analysis: Gap analysis results
        - quality_gates: Pass/fail status for quality gates

    Example:
        >>> packs = build_all_signal_packs()
        >>> metrics = {pa: compute_signal_quality_metrics(pack, pa) for pa, pack in
packs.items()}
        >>> report = generate_quality_report(metrics)
        >>> print(json.dumps(report["summary"], indent=2))
    """
    # Overall statistics
    total_patterns = sum(m.pattern_count for m in metrics_by_pa.values())
    total_indicators = sum(m.indicator_count for m in metrics_by_pa.values())
    total_entities = sum(m.entity_count for m in metrics_by_pa.values())

    avg_confidence = sum(m.threshold_min_confidence for m in metrics_by_pa.values()) /
len(metrics_by_pa)
    avg_evidence = sum(m.threshold_min_evidence for m in metrics_by_pa.values()) /
len(metrics_by_pa)

    high_quality_pas = [
        pa for pa, m in metrics_by_pa.items() if m.is_high_quality
    ]

    # Coverage tier distribution
    tier_distribution = {}
    for m in metrics_by_pa.values():
        tier = m.coverage_tier
        tier_distribution[tier] = tier_distribution.get(tier, 0) + 1

    # Coverage gap analysis
    gap_analysis = analyze_coverage_gaps(metrics_by_pa)

    # Quality gates
    quality_gates = {
        "all_pas_have_patterns": all(m.pattern_count > 0 for m in metrics_by_pa.values()),
        "all_pas_high_quality": len(high_quality_pas) == len(metrics_by_pa),
        "no_critical_gaps": gap_analysis.gap_severity not in ("CRITICAL",),
```

```
        "thresholds_calibrated": abs(gap_analysis.threshold_delta) < 0.10,
    }

    quality_gates["all_gates_passed"] = all(quality_gates.values())

    report = {
        "summary": {
            "total_policy_areas": len(metrics_by_pa),
            "total_patterns": total_patterns,
            "total_indicators": total_indicators,
            "total_entities": total_entities,
            "avg_patterns_per_pa": total_patterns / len(metrics_by_pa),
            "avg_confidence_threshold": round(avg_confidence, 3),
            "avg_evidence_threshold": round(avg_evidence, 3),
            "high_quality_pas": high_quality_pas,
            "high_quality_percentage": round(len(high_quality_pas) / len(metrics_by_pa) *
100, 1),
            "coverage_tier_distribution": tier_distribution,
        },
        "by_policy_area": {
            pa: {
                "pattern_count": m.pattern_count,
                "indicator_count": m.indicator_count,
                "entity_count": m.entity_count,
                "coverage_tier": m.coverage_tier,
                "is_high_quality": m.is_high_quality,
                "threshold_min_confidence": m.threshold_min_confidence,
                "threshold_min_evidence": m.threshold_min_evidence,
                "entity_coverage_ratio": round(m.entity_coverage_ratio, 3),
            }
            for pa, m in metrics_by_pa.items()
        },
        "coverage_gap_analysis": {
            "high_coverage_pas": gap_analysis.high_coverage_pas,
            "low_coverage_pas": gap_analysis.low_coverage_pas,
            "coverage_delta": round(gap_analysis.coverage_delta, 2),
            "threshold_delta": round(gap_analysis.threshold_delta, 3),
            "gap_severity": gap_analysis.gap_severity,
            "requires_fallback_fusion": gap_analysis.requires_fallback_fusion,
            "recommendations": gap_analysis.recommendations,
        },
        "quality_gates": quality_gates,
    }

    logger.info(
        "quality_report_generated",
        total_pas=len(metrics_by_pa),
        all_gates_passed=quality_gates["all_gates_passed"],
        gap_severity=gap_analysis.gap_severity,
    )

    return report

===== FILE: src/saaaaaa/core/orchestrator/signal_registry.py =====
"""
Questionnaire Signal Registry - SOTA Implementation
=====================================================

Content-addressed, type-safe, observable signal registry with cryptographic
consumption tracking and lazy loading.

Technical Standards:
- Pydantic v2 for runtime validation
- OpenTelemetry for distributed tracing
- BLAKE3 for cryptographic hashing
- structlog for structured logging
- Type hints with strict mypy compliance
```

```
Version: 1.0.0
Status: Production-ready
"""

from __future__ import annotations

import hashlib
import time
from collections import defaultdict
from functools import lru_cache
from typing import TYPE_CHECKING, Any, Literal

try:
    import blake3

    BLAKE3_AVAILABLE = True
except ImportError:
    BLAKE3_AVAILABLE = False

try:
    from opentelemetry import trace

    tracer = trace.get_tracer(__name__)
    OTEL_AVAILABLE = True
except ImportError:
    OTEL_AVAILABLE = False
    # Dummy tracer
    class DummySpan:
        def set_attribute(self, key: str, value: Any) -> None:
            pass

        def set_status(self, status: Any) -> None:
            pass

        def record_exception(self, exc: Exception) -> None:
            pass

        def __enter__(self) -> DummySpan:
            return self

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

    class DummyTracer:
        def start_as_current_span(
            self, name: str, attributes: dict[str, Any] | None = None
        ) -> DummySpan:
            return DummySpan()

    tracer = DummyTracer()  # type: ignore

try:
    import structlog

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

    logger = logging.getLogger(__name__)  # type: ignore

from pydantic import BaseModel, ConfigDict, Field, field_validator

if TYPE_CHECKING:
    from .questionnaire import CanonicalQuestionnaire

# ============================================================================
# TYPE-SAFE SIGNAL PACKS (Pydantic v2)
# ============================================================================
```

```python
class ChunkingSignalPack(BaseModel):
    """Type-safe signal pack for Smart Policy Chunking.

    Attributes:
        section_detection_patterns: Regex patterns per PDM section type
        section_weights: Calibrated weights per section (0.0-2.0 range)
        table_patterns: Patterns to detect table boundaries
        numerical_patterns: Patterns to detect numerical content
        embedding_config: Semantic embedding configuration
        version: Signal pack version
        source_hash: Content hash for cache invalidation
    """

    model_config = ConfigDict(frozen=True, strict=True, extra="forbid")

    section_detection_patterns: dict[str, list[str]] = Field(
        ..., min_length=1, description="Patterns per PDM section"
    )
    section_weights: dict[str, float] = Field(
        ..., description="Calibrated weights per section"
    )
    table_patterns: list[str] = Field(
        default_factory=list, description="Table boundary patterns"
    )
    numerical_patterns: list[str] = Field(
        default_factory=list, description="Numerical content patterns"
    )
    embedding_config: dict[str, Any] = Field(
        default_factory=dict, description="Embedding strategy config"
    )
    version: str = Field(default="1.0.0", pattern=r"^\d+\.\d+\.\d+$")
    source_hash: str = Field(..., min_length=32, max_length=64)

    @field_validator("section_weights")
    @classmethod
    def validate_weights(cls, v: dict[str, float]) -> dict[str, float]:
        """Validate section weights are in valid range."""
        for key, weight in v.items():
            if not 0.0 <= weight <= 2.0:
                raise ValueError(f"Weight {key}={weight} out of range [0.0, 2.0]")
        return v


class PatternItem(BaseModel):
    """Individual pattern with metadata."""

    model_config = ConfigDict(frozen=True)

    id: str = Field(..., pattern=r"^PAT-Q\d{3}-\d{3}$")
    pattern: str = Field(..., min_length=1)
    match_type: Literal["REGEX", "LITERAL"]
    confidence_weight: float = Field(..., ge=0.0, le=1.0)
    category: Literal[
        "GENERAL",
        "TEMPORAL",
        "INDICADOR",
        "FUENTE_OFICIAL",
        "TERRITORIAL",
        "UNIDAD_MEDIDA",
    ]
    flags: str = Field(default="", pattern=r"^[imsx]*$")


class ExpectedElement(BaseModel):
    """Expected element specification."""
```

```python
    model_config = ConfigDict(frozen=True)

    type: str = Field(..., min_length=1)
    required: bool = Field(default=False)
    minimum: int = Field(default=0, ge=0)


class MicroAnsweringSignalPack(BaseModel):
    """Type-safe signal pack for Micro Answering."""

    model_config = ConfigDict(frozen=True, strict=True, extra="forbid")

    question_patterns: dict[str, list[PatternItem]] = Field(
        ..., description="Patterns per question ID"
    )
    expected_elements: dict[str, list[ExpectedElement]] = Field(
        ..., description="Expected elements per question"
    )
    indicators_by_pa: dict[str, list[str]] = Field(
        default_factory=dict, description="Indicators per policy area"
    )
    official_sources: list[str] = Field(
        default_factory=list, description="Recognized official sources"
    )
    pattern_weights: dict[str, float] = Field(
        default_factory=dict, description="Confidence weights per pattern ID"
    )
    version: str = Field(default="1.0.0", pattern=r"^\d+\.\d+\.\d+$")
    source_hash: str = Field(..., min_length=32, max_length=64)


class ValidationCheck(BaseModel):
    """Validation check specification."""

    model_config = ConfigDict(frozen=True)

    patterns: list[str] = Field(default_factory=list)
    minimum_required: int = Field(default=1, ge=0)
    minimum_years: int = Field(default=0, ge=0)
    specificity: Literal["HIGH", "MEDIUM", "LOW"] = Field(default="MEDIUM")


class FailureContract(BaseModel):
    """Failure contract specification."""

    model_config = ConfigDict(frozen=True)

    abort_if: list[str] = Field(..., min_length=1)
    emit_code: str = Field(..., pattern=r"^ABORT-Q\d{3}-[A-Z]+$")


class ValidationSignalPack(BaseModel):
    """Type-safe signal pack for Response Validation."""

    model_config = ConfigDict(frozen=True, strict=True, extra="forbid")

    validation_rules: dict[str, dict[str, ValidationCheck]] = Field(
        ..., description="Validation rules per question"
    )
    failure_contracts: dict[str, FailureContract] = Field(
        ..., description="Failure contracts per question"
    )
    modality_thresholds: dict[str, float] = Field(
        default_factory=dict, description="Thresholds per scoring modality"
    )
    abort_codes: dict[str, str] = Field(
        default_factory=dict, description="Abort codes per question"
    )
```

```python
    verification_patterns: dict[str, list[str]] = Field(
        default_factory=dict, description="Verification patterns per question"
    )
    version: str = Field(default="1.0.0", pattern=r"^\d+\.\d+\.\d+$")
    source_hash: str = Field(..., min_length=32, max_length=64)


class AssemblySignalPack(BaseModel):
    """Type-safe signal pack for Response Assembly."""

    model_config = ConfigDict(frozen=True, strict=True, extra="forbid")

    aggregation_methods: dict[str, str] = Field(
        ..., description="Aggregation method per cluster/level"
    )
    cluster_policy_areas: dict[str, list[str]] = Field(
        ..., description="Policy areas per cluster"
    )
    dimension_weights: dict[str, float] = Field(
        default_factory=dict, description="Weights per dimension"
    )
    evidence_keys_by_pa: dict[str, list[str]] = Field(
        default_factory=dict, description="Required evidence keys per policy area"
    )
    coherence_patterns: list[dict[str, Any]] = Field(
        default_factory=list, description="Cross-reference coherence patterns"
    )
    fallback_patterns: dict[str, dict[str, Any]] = Field(
        default_factory=dict, description="Fallback patterns per level"
    )
    version: str = Field(default="1.0.0", pattern=r"^\d+\.\d+\.\d+$")
    source_hash: str = Field(..., min_length=32, max_length=64)


class ModalityConfig(BaseModel):
    """Scoring modality configuration."""

    model_config = ConfigDict(frozen=True)

    aggregation: Literal[
        "presence_threshold",
        "binary_sum",
        "weighted_sum",
        "binary_presence",
        "normalized_continuous",
    ]
    description: str = Field(..., min_length=5)
    failure_code: str = Field(..., pattern=r"^F-[A-F]-[A-Z]+$")
    threshold: float | None = Field(default=None, ge=0.0, le=1.0)
    max_score: int = Field(default=3, ge=0, le=10)
    weights: list[float] | None = Field(default=None)

    @field_validator("weights")
    @classmethod
    def validate_weights_sum(cls, v: list[float] | None) -> list[float] | None:
        """Validate weights sum to 1.0."""
        if v is not None:
            total = sum(v)
            if not 0.99 <= total <= 1.01:  # Allow small floating point error
                raise ValueError(f"Weights must sum to 1.0, got {total}")
        return v


class QualityLevel(BaseModel):
    """Quality level specification."""

    model_config = ConfigDict(frozen=True)
```

```python
    level: Literal["EXCELENTE", "BUENO", "ACEPTABLE", "INSUFICIENTE"]
    min_score: float = Field(..., ge=0.0, le=1.0)
    color: Literal["green", "blue", "yellow", "red"]


class ScoringSignalPack(BaseModel):
    """Type-safe signal pack for Scoring."""

    model_config = ConfigDict(frozen=True, strict=True, extra="forbid")

    question_modalities: dict[str, str] = Field(
        ..., description="Scoring modality per question"
    )
    modality_configs: dict[str, ModalityConfig] = Field(
        ..., description="Configuration per modality type"
    )
    quality_levels: list[QualityLevel] = Field(
        ..., min_length=4, max_length=4, description="Quality level definitions"
    )
    failure_codes: dict[str, str] = Field(
        default_factory=dict, description="Failure codes per modality"
    )
    thresholds: dict[str, float] = Field(
        default_factory=dict, description="Thresholds per modality"
    )
    type_d_weights: list[float] = Field(
        default=[0.4, 0.3, 0.3], description="Weights for TYPE_D modality"
    )
    version: str = Field(default="1.0.0", pattern=r"^\d+\.\d+\.\d+$")
    source_hash: str = Field(..., min_length=32, max_length=64)


# ============================================================================
# CONTENT-ADDRESSED SIGNAL REGISTRY
# ============================================================================


class QuestionnaireSignalRegistry:
    """Content-addressed, observable signal registry with lazy loading.

    Features:
    - Content-based cache invalidation (hash-based)
    - Lazy loading with on-demand materialization
    - OpenTelemetry distributed tracing
    - Structured logging with contextual metadata
    - Type-safe signal packs (Pydantic v2)
    - LRU caching for hot paths

    Architecture:
        CanonicalQuestionnaire → Registry → SignalPacks → Components

    Thread Safety: Single-threaded (use locks for multi-threaded)
    """

    def __init__(self, questionnaire: CanonicalQuestionnaire) -> None:
        """Initialize signal registry.

        Args:
            questionnaire: Canonical questionnaire instance
        """
        self._questionnaire = questionnaire
        self._source_hash = self._compute_source_hash()
        self._initialized = False

        # Lazy-loaded caches
        self._chunking_signals: ChunkingSignalPack | None = None
        self._micro_answering_cache: dict[str, MicroAnsweringSignalPack] = {}
        self._validation_cache: dict[str, ValidationSignalPack] = {}
```

```
        self._assembly_cache: dict[str, AssemblySignalPack] = {}
        self._scoring_cache: dict[str, ScoringSignalPack] = {}

        # Metrics
        self._cache_hits = 0
        self._cache_misses = 0
        self._signal_loads = 0

        logger.info(
            "signal_registry_initialized",
            source_hash=self._source_hash[:16],
            questionnaire_version=questionnaire.version,
        )

    def _compute_source_hash(self) -> str:
        """Compute content hash for cache invalidation."""
        content = str(self._questionnaire.sha256)
        if BLAKE3_AVAILABLE:
            return blake3.blake3(content.encode()).hexdigest()
        else:
            return hashlib.sha256(content.encode()).hexdigest()

    # ========================================================================
    # PUBLIC API: Signal Pack Getters
    # ========================================================================

    def get_chunking_signals(self) -> ChunkingSignalPack:
        """Get signals for Smart Policy Chunking.

        Returns:
            ChunkingSignalPack with section patterns, weights, and config

        Raises:
            ValueError: If signal extraction fails
        """
        with tracer.start_as_current_span(
            "signal_registry.get_chunking_signals",
            attributes={"signal_type": "chunking"},
        ) as span:
            try:
                if self._chunking_signals is None:
                    self._signal_loads += 1
                    self._cache_misses += 1
                    self._chunking_signals = self._build_chunking_signals()
                    span.set_attribute("cache_hit", False)
                else:
                    self._cache_hits += 1
                    span.set_attribute("cache_hit", True)

                span.set_attribute("pattern_count",
len(self._chunking_signals.section_detection_patterns))
                return self._chunking_signals

            except Exception as e:
                span.record_exception(e)
                logger.error("chunking_signals_failed", error=str(e))
                raise

    def get_micro_answering_signals(
        self, question_id: str
    ) -> MicroAnsweringSignalPack:
        """Get signals for Micro Answering for specific question.

        Args:
            question_id: Question ID (Q001-Q300)

        Returns:
            MicroAnsweringSignalPack with patterns, elements, indicators
```

```
        Raises:
            ValueError: If question not found or signal extraction fails
        """
        with tracer.start_as_current_span(
            "signal_registry.get_micro_answering_signals",
            attributes={"signal_type": "micro_answering", "question_id": question_id},
        ) as span:
            try:
                if question_id in self._micro_answering_cache:
                    self._cache_hits += 1
                    span.set_attribute("cache_hit", True)
                    return self._micro_answering_cache[question_id]

                self._signal_loads += 1
                self._cache_misses += 1
                span.set_attribute("cache_hit", False)

                pack = self._build_micro_answering_signals(question_id)
                self._micro_answering_cache[question_id] = pack

                span.set_attribute("pattern_count",
len(pack.question_patterns.get(question_id, [])))
                return pack

            except Exception as e:
                span.record_exception(e)
                logger.error(
                    "micro_answering_signals_failed", question_id=question_id,
error=str(e)
                )
                raise

    def get_validation_signals(self, question_id: str) -> ValidationSignalPack:
        """Get signals for Response Validation for specific question.

        Args:
            question_id: Question ID (Q001-Q300)

        Returns:
            ValidationSignalPack with rules, contracts, thresholds

        Raises:
            ValueError: If question not found or signal extraction fails
        """
        with tracer.start_as_current_span(
            "signal_registry.get_validation_signals",
            attributes={"signal_type": "validation", "question_id": question_id},
        ) as span:
            try:
                if question_id in self._validation_cache:
                    self._cache_hits += 1
                    span.set_attribute("cache_hit", True)
                    return self._validation_cache[question_id]

                self._signal_loads += 1
                self._cache_misses += 1
                span.set_attribute("cache_hit", False)

                pack = self._build_validation_signals(question_id)
                self._validation_cache[question_id] = pack

                span.set_attribute("rule_count",
len(pack.validation_rules.get(question_id, {})))
                return pack

            except Exception as e:
                span.record_exception(e)
```

```python
            logger.error(
                "validation_signals_failed", question_id=question_id, error=str(e)
            )
            raise

    def get_assembly_signals(self, level: str) -> AssemblySignalPack:
        """Get signals for Response Assembly at specified level.

        Args:
            level: Assembly level (MESO_1, MESO_2, etc. or MACRO_1)

        Returns:
            AssemblySignalPack with aggregation methods, clusters, weights

        Raises:
            ValueError: If level not found or signal extraction fails
        """
        with tracer.start_as_current_span(
            "signal_registry.get_assembly_signals",
            attributes={"signal_type": "assembly", "level": level},
        ) as span:
            try:
                if level in self._assembly_cache:
                    self._cache_hits += 1
                    span.set_attribute("cache_hit", True)
                    return self._assembly_cache[level]

                self._signal_loads += 1
                self._cache_misses += 1
                span.set_attribute("cache_hit", False)

                pack = self._build_assembly_signals(level)
                self._assembly_cache[level] = pack

                span.set_attribute("cluster_count", len(pack.cluster_policy_areas))
                return pack

            except Exception as e:
                span.record_exception(e)
                logger.error("assembly_signals_failed", level=level, error=str(e))
                raise

    def get_scoring_signals(self, question_id: str) -> ScoringSignalPack:
        """Get signals for Scoring for specific question.

        Args:
            question_id: Question ID (Q001-Q300)

        Returns:
            ScoringSignalPack with modalities, configs, quality levels

        Raises:
            ValueError: If question not found or signal extraction fails
        """
        with tracer.start_as_current_span(
            "signal_registry.get_scoring_signals",
            attributes={"signal_type": "scoring", "question_id": question_id},
        ) as span:
            try:
                if question_id in self._scoring_cache:
                    self._cache_hits += 1
                    span.set_attribute("cache_hit", True)
                    return self._scoring_cache[question_id]

                self._signal_loads += 1
                self._cache_misses += 1
                span.set_attribute("cache_hit", False)
```

```python
            pack = self._build_scoring_signals(question_id)
            self._scoring_cache[question_id] = pack

            modality = pack.question_modalities.get(question_id, "UNKNOWN")
            span.set_attribute("modality", modality)
            return pack

        except Exception as e:
            span.record_exception(e)
            logger.error("scoring_signals_failed", question_id=question_id,
error=str(e))
            raise

    # ============================================================================
    # PRIVATE: Signal Pack Builders
    # ============================================================================

    def _build_chunking_signals(self) -> ChunkingSignalPack:
        """Build chunking signal pack from questionnaire."""
        blocks = dict(self._questionnaire.data.get("blocks", {}))
        semantic_layers = blocks.get("semantic_layers", {})

        # Extract section patterns (from micro questions)
        section_patterns: dict[str, list[str]] = defaultdict(list)
        micro_questions = blocks.get("micro_questions", [])

        for q in micro_questions:
            for pattern_obj in q.get("patterns", []):
                category = pattern_obj.get("category", "GENERAL")
                pattern = pattern_obj.get("pattern", "")
                if pattern:
                    section_patterns[category].append(pattern)

        # Deduplicate
        section_patterns = {k: list(set(v)) for k, v in section_patterns.items()}

        # Section weights (hardcoded calibrated values for now)
        section_weights = {
            "DIAGNOSTICO": 0.92,
            "PLAN_INVERSIONES": 1.25,
            "PLAN_PLURIANUAL": 1.18,
            "VISION_ESTRATEGICA": 1.0,
            "MARCO_FISCAL": 1.0,
            "SEGUIMIENTO": 1.0,
        }

        # Table patterns
        table_patterns = [
            r"\|.*\|.*\|",  # Markdown table
            r"<table",  # HTML table
            r"Cuadro \d+",  # Spanish table reference
            r"Tabla \d+",
        ]

        # Numerical patterns
        numerical_patterns = [
            r"\d+%",  # Percentage
            r"\$\s*\d+",  # Currency
            r"\d+\.\d+",  # Decimal
            r"\d+,\d+",  # Decimal (Spanish)
        ]

        return ChunkingSignalPack(
            section_detection_patterns=section_patterns,
            section_weights=section_weights,
            table_patterns=table_patterns,
            numerical_patterns=numerical_patterns,
            embedding_config=semantic_layers.get("embedding_strategy", {}),
```

```python
            source_hash=self._source_hash,
        )

    def _build_micro_answering_signals(
        self, question_id: str
    ) -> MicroAnsweringSignalPack:
        """Build micro answering signal pack for question."""
        question = self._get_question(question_id)

        # Extract patterns
        patterns_raw = question.get("patterns", [])
        patterns = [
            PatternItem(
                id=p.get("id", f"PAT-{question_id}-000"),
                pattern=p.get("pattern", ""),
                match_type=p.get("match_type", "REGEX"),
                confidence_weight=p.get("confidence_weight", 0.85),
                category=p.get("category", "GENERAL"),
                flags=p.get("flags", ""),
            )
            for p in patterns_raw
        ]

        # Extract expected elements
        elements_raw = question.get("expected_elements", [])
        elements = [
            ExpectedElement(
                type=e.get("type", "unknown"),
                required=e.get("required", False),
                minimum=e.get("minimum", 0),
            )
            for e in elements_raw
        ]

        # Get indicators by policy area
        pa = question.get("policy_area_id", "PA01")
        indicators = self._extract_indicators_for_pa(pa)

        # Get official sources
        official_sources = self._extract_official_sources()

        # Pattern weights
        pattern_weights = {
            p.id: p.confidence_weight for p in patterns
        }

        return MicroAnsweringSignalPack(
            question_patterns={question_id: patterns},
            expected_elements={question_id: elements},
            indicators_by_pa={pa: indicators},
            official_sources=official_sources,
            pattern_weights=pattern_weights,
            source_hash=self._source_hash,
        )

    def _build_validation_signals(self, question_id: str) -> ValidationSignalPack:
        """Build validation signal pack for question."""
        question = self._get_question(question_id)
        blocks = dict(self._questionnaire.data.get("blocks", {}))
        scoring = blocks.get("scoring", {})

        # Extract validation rules
        validations_raw = question.get("validations", {})
        validation_rules = {}
        for rule_name, rule_data in validations_raw.items():
            validation_rules[rule_name] = ValidationCheck(
                patterns=rule_data.get("patterns", []),
                minimum_required=rule_data.get("minimum_required", 1),
```

```
            minimum_years=rule_data.get("minimum_years", 0),
            specificity=rule_data.get("specificity", "MEDIUM"),
        )

    # Extract failure contract
    failure_contract_raw = question.get("failure_contract", {})
    failure_contract = None
    if failure_contract_raw:
        failure_contract = FailureContract(
            abort_if=failure_contract_raw.get("abort_if",
["missing_required_element"]),
            emit_code=failure_contract_raw.get("emit_code",
f"ABORT-{question_id}-REQ"),
        )

    # Get modality thresholds
    modality_definitions = scoring.get("modality_definitions", {})
    modality_thresholds = {
        k: v.get("threshold", 0.7)
        for k, v in modality_definitions.items()
        if "threshold" in v
    }

    return ValidationSignalPack(
        validation_rules={question_id: validation_rules} if validation_rules else {},
        failure_contracts={question_id: failure_contract} if failure_contract else {},
        modality_thresholds=modality_thresholds,
        abort_codes={question_id: failure_contract.emit_code} if failure_contract else
 {},
        verification_patterns={question_id: list(validation_rules.keys())},
        source_hash=self._source_hash,
    )

  def _build_assembly_signals(self, level: str) -> AssemblySignalPack:
    """Build assembly signal pack for level."""
    blocks = dict(self._questionnaire.data.get("blocks", {}))
    niveles = blocks.get("niveles_abstraccion", {})

    # Extract aggregation methods
    aggregation_methods = {}
    if level.startswith("MESO"):
        meso_questions = blocks.get("meso_questions", [])
        for meso_q in meso_questions:
            agg_method = meso_q.get("aggregation_method", "weighted_average")
            q_id = meso_q.get("question_id", "UNKNOWN")
            aggregation_methods[q_id] = agg_method
    else:
        macro_q = blocks.get("macro_question", {})
        agg_method = macro_q.get("aggregation_method", "holistic_assessment")
        aggregation_methods["MACRO_1"] = agg_method

    # Extract cluster composition
    clusters = niveles.get("clusters", [])
    cluster_policy_areas = {
        c.get("cluster_id", "UNKNOWN"): c.get("policy_area_ids", [])
        for c in clusters
    }

    # Dimension weights (uniform for now)
    dimension_weights = {
        f"DIM{i:02d}": 1.0 / 6 for i in range(1, 7)
    }

    # Evidence keys by policy area
    policy_areas = niveles.get("policy_areas", [])
    evidence_keys_by_pa = {
        pa.get("policy_area_id", "UNKNOWN"): pa.get("required_evidence_keys", [])
        for pa in policy_areas
```

```python
    }

    # Coherence patterns (from meso questions)
    coherence_patterns = []
    meso_questions = blocks.get("meso_questions", [])
    for meso_q in meso_questions:
        patterns = meso_q.get("patterns", [])
        coherence_patterns.extend(patterns)

    # Fallback patterns
    fallback_patterns = {}
    macro_q = blocks.get("macro_question", {})
    if "fallback" in macro_q:
        fallback_patterns["MACRO_1"] = macro_q["fallback"]

    return AssemblySignalPack(
        aggregation_methods=aggregation_methods,
        cluster_policy_areas=cluster_policy_areas,
        dimension_weights=dimension_weights,
        evidence_keys_by_pa=evidence_keys_by_pa,
        coherence_patterns=coherence_patterns,
        fallback_patterns=fallback_patterns,
        source_hash=self._source_hash,
    )

def _build_scoring_signals(self, question_id: str) -> ScoringSignalPack:
    """Build scoring signal pack for question."""
    question = self._get_question(question_id)
    blocks = dict(self._questionnaire.data.get("blocks", {}))
    scoring = blocks.get("scoring", {})

    # Get question modality
    modality = question.get("scoring_modality", "TYPE_A")

    # Extract modality configs
    modality_definitions = scoring.get("modality_definitions", {})
    modality_configs = {}
    for mod_type, mod_def in modality_definitions.items():
        modality_configs[mod_type] = ModalityConfig(
            aggregation=mod_def.get("aggregation", "presence_threshold"),
            description=mod_def.get("description", ""),
            failure_code=mod_def.get("failure_code", f"F-{mod_type[-1]}-MIN"),
            threshold=mod_def.get("threshold"),
            max_score=mod_def.get("max_score", 3),
            weights=mod_def.get("weights"),
        )

    # Extract quality levels
    micro_levels = scoring.get("micro_levels", [])
    quality_levels = [
        QualityLevel(
            level=lvl.get("level", "INSUFICIENTE"),
            min_score=lvl.get("min_score", 0.0),
            color=lvl.get("color", "red"),
        )
        for lvl in micro_levels
    ]

    # Failure codes
    failure_codes = {
        k: v.get("failure_code", f"F-{k[-1]}-MIN")
        for k, v in modality_definitions.items()
    }

    # Thresholds
    thresholds = {
        k: v.get("threshold", 0.7)
        for k, v in modality_definitions.items()
```

```
            if "threshold" in v
        }

        # TYPE_D weights
        type_d_weights = modality_definitions.get("TYPE_D", {}).get("weights", [0.4, 0.3,
0.3])

        return ScoringSignalPack(
            question_modalities={question_id: modality},
            modality_configs=modality_configs,
            quality_levels=quality_levels,
            failure_codes=failure_codes,
            thresholds=thresholds,
            type_d_weights=type_d_weights,
            source_hash=self._source_hash,
        )

    # ============================================================================
    # HELPER METHODS
    # ============================================================================

    def _get_question(self, question_id: str) -> dict[str, Any]:
        """Get question by ID from questionnaire."""
        for q in self._questionnaire.micro_questions:
            if dict(q).get("question_id") == question_id:
                return dict(q)
        raise ValueError(f"Question {question_id} not found in questionnaire")

    def _extract_indicators_for_pa(self, policy_area: str) -> list[str]:
        """Extract indicator patterns for policy area."""
        indicators = []
        blocks = dict(self._questionnaire.data.get("blocks", {}))
        micro_questions = blocks.get("micro_questions", [])

        for q in micro_questions:
            if q.get("policy_area_id") == policy_area:
                for pattern_obj in q.get("patterns", []):
                    if pattern_obj.get("category") == "INDICADOR":
                        indicators.append(pattern_obj.get("pattern", ""))

        return list(set(indicators))

    def _extract_official_sources(self) -> list[str]:
        """Extract official source patterns from all questions."""
        sources = []
        blocks = dict(self._questionnaire.data.get("blocks", {}))
        micro_questions = blocks.get("micro_questions", [])

        for q in micro_questions:
            for pattern_obj in q.get("patterns", []):
                if pattern_obj.get("category") == "FUENTE_OFICIAL":
                    pattern = pattern_obj.get("pattern", "")
                    # Split on | for multiple sources in one pattern
                    sources.extend(p.strip() for p in pattern.split("|") if p.strip())

        return list(set(sources))

    # ============================================================================
    # OBSERVABILITY
    # ============================================================================

    def get_metrics(self) -> dict[str, Any]:
        """Get registry metrics for observability.

        Returns:
            Dictionary with cache hits, misses, signal loads, etc.
        """
        total_requests = self._cache_hits + self._cache_misses
```

```python
        hit_rate = self._cache_hits / total_requests if total_requests > 0 else 0.0

        return {
            "cache_hits": self._cache_hits,
            "cache_misses": self._cache_misses,
            "hit_rate": hit_rate,
            "signal_loads": self._signal_loads,
            "cached_micro_answering": len(self._micro_answering_cache),
            "cached_validation": len(self._validation_cache),
            "cached_assembly": len(self._assembly_cache),
            "cached_scoring": len(self._scoring_cache),
            "source_hash": self._source_hash[:16],
        }

    def clear_cache(self) -> None:
        """Clear all caches (for testing or hot-reload)."""
        self._chunking_signals = None
        self._micro_answering_cache.clear()
        self._validation_cache.clear()
        self._assembly_cache.clear()
        self._scoring_cache.clear()

        logger.info("signal_registry_cache_cleared")


# ============================================================================
# FACTORY INTEGRATION
# ============================================================================


def create_signal_registry(
    questionnaire: CanonicalQuestionnaire,
) -> QuestionnaireSignalRegistry:
    """Factory function to create signal registry.

    Args:
        questionnaire: Canonical questionnaire instance

    Returns:
        Initialized signal registry

    Example:
        >>> from saaaaaa.core.orchestrator.questionnaire import load_questionnaire
        >>> canonical = load_questionnaire()
        >>> registry = create_signal_registry(canonical)
        >>> signals = registry.get_chunking_signals()
    """
    return QuestionnaireSignalRegistry(questionnaire)


__all__ = [
    "QuestionnaireSignalRegistry",
    "ChunkingSignalPack",
    "MicroAnsweringSignalPack",
    "ValidationSignalPack",
    "AssemblySignalPack",
    "ScoringSignalPack",
    "PatternItem",
    "ExpectedElement",
    "ValidationCheck",
    "FailureContract",
    "ModalityConfig",
    "QualityLevel",
    "create_signal_registry",
]
```

===== FILE: src/saaaaaa/core/orchestrator/signals.py =====

```python
"""Cross-Cut Signal Channel: questionnaire.monolith → orchestrator.
```

```
This module implements the strategic signal propagation system that continuously
irrigates patterns, indicators, regex, verbs, entities, and thresholds into the
answer-generation process.

Architecture:
- SignalPack: Typed, versioned signal payload
- SignalRegistry: In-memory LRU cache with TTL
- SignalClient: Circuit-breaker enabled HTTP client
- Signal-aware execution integration

Design Principles:
- Deterministic signal application
- Graceful degradation on signal unavailability
- Full traceability of signal usage
- Observability via metrics and structured logging
"""

from __future__ import annotations

import json
import time
from collections import OrderedDict
from dataclasses import dataclass, field
from datetime import datetime, timezone
from typing import Any, Literal

# Optional dependency - blake3
try:
    import blake3
    BLAKE3_AVAILABLE = True
except ImportError:
    BLAKE3_AVAILABLE = False
    import hashlib
    # Fallback to hashlib if blake3 not available
    class blake3:  # type: ignore
        @staticmethod
        def blake3(data: bytes) -> object:
            class HashResult:
                def __init__(self, data: bytes) -> None:
                    self._hash = hashlib.sha256(data)
                def hexdigest(self) -> str:
                    return self._hash.hexdigest()
            return HashResult(data)
# Optional dependency - structlog
try:
    import structlog
    STRUCTLOG_AVAILABLE = True
except ImportError:
    STRUCTLOG_AVAILABLE = False
    import logging
    structlog = logging  # type: ignore  # Fallback to standard logging
from pydantic import BaseModel, Field, field_validator

# Optional dependency - tenacity
try:
    from tenacity import (
        retry,
        retry_if_exception_type,
        stop_after_attempt,
        wait_exponential,
    )
    TENACITY_AVAILABLE = True
except ImportError:
    TENACITY_AVAILABLE = False
    # Dummy decorator when tenacity not available
    def retry(*args, **kwargs):  # type: ignore
        def decorator(func):
```

```
            return func
        return decorator
    def stop_after_attempt(x) -> None:
        return None  # type: ignore
    def wait_exponential(**kwargs) -> None:
        return None  # type: ignore
    def retry_if_exception_type(x) -> None:
        return None  # type: ignore


logger = structlog.get_logger(__name__) if STRUCTLOG_AVAILABLE else
logging.getLogger(__name__)


PolicyArea = Literal[
    "PA01", "PA02", "PA03", "PA04", "PA05",
    "PA06", "PA07", "PA08", "PA09", "PA10",
    # Legacy policy areas (kept for backward compatibility)
    "fiscal", "salud", "ambiente", "energía", "transporte"
]


class SignalPack(BaseModel):
    """
    Versioned strategic signal payload for policy-aware execution.

    Contains curated patterns, indicators, and thresholds specific to a policy area.
    All packs carry fingerprints for drift detection and validation windows.

    Attributes:
        version: Semantic version string (e.g., "1.0.0")
        policy_area: Policy domain this pack targets
        patterns: Text patterns for narrative detection
        indicators: Key performance indicators for scoring
        regex: Regular expressions for structured extraction
        verbs: Action verbs for policy intent detection
        entities: Named entities relevant to policy area
        thresholds: Named thresholds for scoring/filtering
        ttl_s: Time-to-live in seconds for cache management
        source_fingerprint: BLAKE3 hash of source content
        valid_from: ISO timestamp when signal becomes valid
        valid_to: ISO timestamp when signal expires
        metadata: Optional additional metadata
    """

    version: str = Field(
        description="Semantic version string (e.g., '1.0.0')"
    )
    policy_area: PolicyArea = Field(
        description="Policy domain this pack targets"
    )
    patterns: list[str] = Field(
        default_factory=list,
        description="Text patterns for narrative detection"
    )
    indicators: list[str] = Field(
        default_factory=list,
        description="Key performance indicators for scoring"
    )
    regex: list[str] = Field(
        default_factory=list,
        description="Regular expressions for structured extraction"
    )
    verbs: list[str] = Field(
        default_factory=list,
        description="Action verbs for policy intent detection"
    )
    entities: list[str] = Field(
```

```
    default_factory=list,
    description="Named entities relevant to policy area"
)
thresholds: dict[str, float] = Field(
    default_factory=dict,
    description="Named thresholds for scoring/filtering"
)
ttl_s: int = Field(
    default=3600,
    ge=0,
    description="Time-to-live in seconds for cache management"
)
source_fingerprint: str = Field(
    default="",
    description="BLAKE3 hash of source content"
)
valid_from: str = Field(
    default_factory=lambda: datetime.now(timezone.utc).isoformat(),
    description="ISO timestamp when signal becomes valid"
)
valid_to: str = Field(
    default="",
    description="ISO timestamp when signal expires"
)
metadata: dict[str, Any] = Field(
    default_factory=dict,
    description="Optional additional metadata"
)

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

@field_validator("version")
@classmethod
def validate_version(cls, v: str) -> str:
    """Validate semantic version format."""
    parts = v.split(".")
    if len(parts) != 3:
        raise ValueError(f"Version must be in format 'X.Y.Z', got '{v}'")
    for part in parts:
        if not part.isdigit():
            raise ValueError(f"Version parts must be numeric, got '{v}'")
    return v

@field_validator("thresholds")
@classmethod
def validate_thresholds(cls, v: dict[str, float]) -> dict[str, float]:
    """Validate threshold values are in valid range."""
    for key, value in v.items():
        if not (0.0 <= value <= 1.0):
            raise ValueError(
                f"Threshold '{key}' must be in range [0.0, 1.0], got {value}"
            )
    return v

def compute_hash(self) -> str:
    """
    Compute deterministic BLAKE3 hash of signal pack content.

    Returns:
        Hex string of BLAKE3 hash
    """
    # Use model_dump to get a dict, then sort keys manually
    content_dict = self.model_dump(
        exclude={"source_fingerprint", "valid_from", "valid_to", "metadata"},
    )
```

```
        # Sort keys for deterministic hashing
        content_json = json.dumps(content_dict, sort_keys=True, separators=(',', ':'))
        return blake3.blake3(content_json.encode("utf-8")).hexdigest()

    @staticmethod
    def _parse_iso_timestamp(timestamp_str: str) -> datetime:
        """
        Parse ISO timestamp with Z suffix to datetime.

        Args:
            timestamp_str: ISO 8601 timestamp string

        Returns:
            Parsed datetime object
        """
        return datetime.fromisoformat(timestamp_str.replace("Z", "+00:00"))

    def is_valid(self, now: datetime | None = None) -> bool:
        """
        Check if signal pack is currently valid.

        Args:
            now: Current time (defaults to utcnow)

        Returns:
            True if signal is within validity window
        """
        if now is None:
            now = datetime.now(timezone.utc)

        valid_from_dt = self._parse_iso_timestamp(self.valid_from)
        if now < valid_from_dt:
            return False

        if self.valid_to:
            valid_to_dt = self._parse_iso_timestamp(self.valid_to)
            if now > valid_to_dt:
                return False

        return True

    def get_keys_used(self) -> list[str]:
        """
        Get list of signal keys that have non-empty values.

        Returns:
            List of key names with content
        """
        keys = []
        if self.patterns:
            keys.append("patterns")
        if self.indicators:
            keys.append("indicators")
        if self.regex:
            keys.append("regex")
        if self.verbs:
            keys.append("verbs")
        if self.entities:
            keys.append("entities")
        if self.thresholds:
            keys.append("thresholds")
        return keys


@dataclass
class CacheEntry:
    """Entry in the signal registry cache."""
```

```
    signal_pack: SignalPack
    inserted_at: float
    access_count: int = 0
    last_accessed: float = field(default_factory=time.time)


class SignalRegistry:
    """
    In-memory LRU cache for signal packs with TTL management.

    Features:
    - LRU eviction when capacity exceeded
    - TTL-based expiration
    - Access tracking for observability
    - Thread-safe operations (single-process)

    Attributes:
        max_size: Maximum number of cached signal packs
        default_ttl_s: Default TTL for cached entries
    """

    def __init__(self, max_size: int = 100, default_ttl_s: int = 3600) -> None:
        """
        Initialize signal registry.

        Args:
            max_size: Maximum cache size
            default_ttl_s: Default TTL in seconds
        """
        self._cache: OrderedDict[str, CacheEntry] = OrderedDict()
        self._max_size = max_size
        self._default_ttl_s = default_ttl_s
        self._hits = 0
        self._misses = 0
        self._evictions = 0

        logger.info(
            "signal_registry_initialized",
            max_size=max_size,
            default_ttl_s=default_ttl_s,
        )

    def put(self, policy_area: str, signal_pack: SignalPack) -> None:
        """
        Store signal pack in registry.

        Args:
            policy_area: Policy area key
            signal_pack: Signal pack to store
        """
        now = time.time()

        # Remove expired entries before insertion
        self._evict_expired()

        # LRU eviction if at capacity
        if len(self._cache) >= self._max_size and policy_area not in self._cache:
            oldest_key = next(iter(self._cache))
            self._cache.pop(oldest_key)
            self._evictions += 1
            logger.debug("signal_registry_evicted_lru", key=oldest_key)

        # Insert or update
        entry = CacheEntry(signal_pack=signal_pack, inserted_at=now)
        self._cache[policy_area] = entry
        self._cache.move_to_end(policy_area)  # Mark as most recently used

        logger.info(
```

```
        "signal_registry_put",
        policy_area=policy_area,
        version=signal_pack.version,
        hash=signal_pack.compute_hash()[:16],
    )

def get(self, policy_area: str) -> SignalPack | None:
    """
    Retrieve signal pack from registry.

    Args:
        policy_area: Policy area key

    Returns:
        Signal pack if found and valid, None otherwise
    """
    now = time.time()

    entry = self._cache.get(policy_area)
    if entry is None:
        self._misses += 1
        logger.debug("signal_registry_miss", policy_area=policy_area)
        return None

    # Check TTL expiration
    ttl = entry.signal_pack.ttl_s or self._default_ttl_s
    if now - entry.inserted_at > ttl:
        # Expired, remove from cache
        self._cache.pop(policy_area)
        self._misses += 1
        logger.debug(
            "signal_registry_expired",
            policy_area=policy_area,
            age_s=now - entry.inserted_at,
        )
        return None

    # Check validity window
    if not entry.signal_pack.is_valid():
        self._cache.pop(policy_area)
        self._misses += 1
        logger.debug("signal_registry_invalid", policy_area=policy_area)
        return None

    # Valid hit
    entry.access_count += 1
    entry.last_accessed = now
    self._cache.move_to_end(policy_area)  # Mark as most recently used
    self._hits += 1

    logger.debug(
        "signal_registry_hit",
        policy_area=policy_area,
        access_count=entry.access_count,
    )

    return entry.signal_pack

def _evict_expired(self) -> None:
    """Remove expired entries from cache."""
    now = time.time()
    expired_keys = []

    for key, entry in self._cache.items():
        ttl = entry.signal_pack.ttl_s or self._default_ttl_s
        if now - entry.inserted_at > ttl:
            expired_keys.append(key)
```

```
        for key in expired_keys:
            self._cache.pop(key)
            self._evictions += 1

        if expired_keys:
            logger.debug("signal_registry_evicted_expired", count=len(expired_keys))

    def get_metrics(self) -> dict[str, Any]:
        """
        Get registry metrics for observability.

        Returns:
            Dict with metrics:
            - hit_rate: Cache hit rate [0.0, 1.0]
            - size: Current cache size
            - capacity: Maximum cache size
            - hits: Total cache hits
            - misses: Total cache misses
            - evictions: Total evictions
        """
        total = self._hits + self._misses
        hit_rate = self._hits / total if total > 0 else 0.0

        # Compute staleness stats
        now = time.time()
        staleness_values = []
        for entry in self._cache.values():
            staleness_values.append(now - entry.inserted_at)

        avg_staleness = sum(staleness_values) / len(staleness_values) if staleness_values
else 0.0
        max_staleness = max(staleness_values) if staleness_values else 0.0

        return {
            "hit_rate": hit_rate,
            "size": len(self._cache),
            "capacity": self._max_size,
            "hits": self._hits,
            "misses": self._misses,
            "evictions": self._evictions,
            "staleness_avg_s": avg_staleness,
            "staleness_max_s": max_staleness,
        }

    def clear(self) -> None:
        """Clear all entries from registry."""
        self._cache.clear()
        logger.info("signal_registry_cleared")


class CircuitBreakerError(Exception):
    """Raised when circuit breaker is open."""
    pass


class SignalUnavailableError(Exception):
    """Raised when signal service is unavailable or returns error."""

    def __init__(self, message: str, status_code: int | None = None) -> None:
        super().__init__(message)
        self.status_code = status_code


class InMemorySignalSource:
    """
    In-memory signal source for local/testing mode.

    Provides signal packs directly from memory without HTTP calls.
```

```python
    Used when base_url starts with "memory://".
    """

    def __init__(self) -> None:
        """Initialize in-memory signal source."""
        self._signals: dict[str, SignalPack] = {}
        logger.info("in_memory_signal_source_initialized")

    def register(self, policy_area: str, signal_pack: SignalPack) -> None:
        """
        Register a signal pack for a policy area.

        Args:
            policy_area: Policy area key
            signal_pack: Signal pack to register
        """
        self._signals[policy_area] = signal_pack
        logger.debug(
            "signal_registered",
            policy_area=policy_area,
            version=signal_pack.version,
        )

    def get(self, policy_area: str) -> SignalPack | None:
        """
        Get signal pack for policy area.

        Args:
            policy_area: Policy area key

        Returns:
            SignalPack if found, None otherwise
        """
        pack = self._signals.get(policy_area)
        if pack:
            logger.debug("memory_signal_hit", policy_area=policy_area)
        else:
            logger.debug("memory_signal_miss", policy_area=policy_area)
        return pack


class SignalClient:
    """
    Signal client supporting both memory:// and HTTP transports.

    Features:
    - memory:// URL scheme for in-process signals (default)
    - HTTP with httpx (behind enable_http_signals flag)
    - ETag support for conditional requests (304 Not Modified)
    - Circuit breaker for fault isolation
    - Automatic retry with exponential backoff
    - Response size validation (≤1.5 MB)
    - Timeout enforcement (≤5s by default)
    - Structured logging and observability

    URL Schemes:
    - memory://: In-process signal source (no network calls)
    - http://...: HTTP signal service with circuit breaker
    - https://...: HTTPS signal service with circuit breaker

    HTTP Status Code Mapping:
    - 200 OK → SignalPack (validated with Pydantic)
    - 304 Not Modified → None (cache is fresh)
    - 401/403 Unauthorized/Forbidden → SignalUnavailableError
    - 429 Too Many Requests → SignalUnavailableError (with retry)
    - 500+ Server Error → SignalUnavailableError (with retry)
    - Timeout → SignalUnavailableError
    """
```

```
# Maximum response size: 1.5 MB
MAX_RESPONSE_SIZE_BYTES = 1_500_000

def __init__(
    self,
    base_url: str = "memory://",
    max_retries: int = 3,
    timeout_s: float = 5.0,
    circuit_breaker_threshold: int = 5,
    circuit_breaker_cooldown_s: float = 60.0,
    enable_http_signals: bool = False,
    memory_source: InMemorySignalSource | None = None,
) -> None:
    """
    Initialize signal client.

    Args:
        base_url: Base URL for signal service or "memory://" for in-process
        max_retries: Maximum retry attempts for HTTP
        timeout_s: Request timeout in seconds (≤5s recommended)
        circuit_breaker_threshold: Failures before circuit opens (default: 5)
        circuit_breaker_cooldown_s: Cooldown period in seconds (default: 60s)
        enable_http_signals: Enable HTTP transport (requires http:// or https:// URL)
        memory_source: InMemorySignalSource for memory:// mode
    """
    self._base_url = base_url.rstrip("/")
    self._max_retries = max_retries
    self._timeout_s = min(timeout_s, 5.0)  # Cap at 5s
    self._circuit_breaker_threshold = circuit_breaker_threshold
    self._circuit_breaker_cooldown_s = circuit_breaker_cooldown_s
    self._enable_http_signals = enable_http_signals

    # Circuit breaker state
    self._failure_count = 0
    self._circuit_open = False
    self._last_failure_time = 0.0
    self._state_changes: list[dict[str, Any]] = []
    self._max_history = 100

    # Determine transport mode
    if base_url.startswith("memory://"):
        self._transport = "memory"
        self._memory_source = memory_source or InMemorySignalSource()
    elif base_url.startswith(("http://", "https://")):
        if not enable_http_signals:
            logger.warning(
                "http_signals_disabled",
                message="HTTP URL provided but enable_http_signals=False. "
                        "Falling back to memory:// mode.",
            )
            self._transport = "memory"
            self._memory_source = memory_source or InMemorySignalSource()
        else:
            self._transport = "http"
            self._memory_source = None
            # Import httpx only when needed
            try:
                import httpx
                self._httpx = httpx
            except ImportError as e:
                raise ImportError(
                    "httpx is required for HTTP signal transport. "
                    "Install with: pip install httpx"
                ) from e
    else:
        raise ValueError(
            f"Invalid base_url scheme: {base_url}. "
```

```
                "Must start with 'memory://', 'http://', or 'https://'"
            )

        # ETag cache for conditional requests
        self._etag_cache: dict[str, str] = {}

        logger.info(
            "signal_client_initialized",
            base_url=base_url,
            transport=self._transport,
            timeout_s=self._timeout_s,
            enable_http_signals=enable_http_signals,
        )

    def fetch_signal_pack(
        self,
        policy_area: str,
        etag: str | None = None,
    ) -> SignalPack | None:
        """
        Fetch signal pack from signal source.

        Args:
            policy_area: Policy area to fetch
            etag: Optional ETag for conditional request (HTTP only)

        Returns:
            SignalPack if successful and fresh
            None if 304 Not Modified or service unavailable

        Raises:
            CircuitBreakerError: If circuit breaker is open
            SignalUnavailableError: If service returns error status
        """
        if self._transport == "memory":
            return self._fetch_from_memory(policy_area)
        else:
            return self._fetch_from_http(policy_area, etag)

    def _fetch_from_memory(self, policy_area: str) -> SignalPack | None:
        """Fetch signal pack from in-memory source."""
        if self._memory_source is None:
            logger.error("memory_source_not_initialized")
            return None

        return self._memory_source.get(policy_area)

    @retry(
        stop=stop_after_attempt(3),
        wait=wait_exponential(multiplier=1, min=1, max=10),
        retry=retry_if_exception_type(ConnectionError),
    )
    def _fetch_from_http(
        self,
        policy_area: str,
        etag: str | None = None,
    ) -> SignalPack | None:
        """Fetch signal pack from HTTP service."""
        # Check circuit breaker
        if self._circuit_open:
            now = time.time()
            if now - self._last_failure_time < self._circuit_breaker_cooldown_s:
                logger.warning(
                    "signal_client_circuit_open",
                    policy_area=policy_area,
                    cooldown_remaining=self._circuit_breaker_cooldown_s - (now -
self._last_failure_time),
                )
```

```
                raise CircuitBreakerError(
                    f"Circuit breaker is open. Cooldown remaining: "
                    f"{self._circuit_breaker_cooldown_s - (now -
self._last_failure_time):.1f}s"
                )
            else:
                # Try to close circuit
                old_open = self._circuit_open
                self._circuit_open = False
                self._failure_count = 0

                # Record state change
                self._state_changes.append({
                    'timestamp': time.time(),
                    'from_open': old_open,
                    'to_open': self._circuit_open,
                    'failures': self._failure_count,
                })

                # Trim history
                if len(self._state_changes) > self._max_history:
                    self._state_changes = self._state_changes[-self._max_history:]

                logger.info("signal_client_circuit_closed")

        # Build request
        url = f"{self._base_url}/signals/{policy_area}"
        headers = {}

        # Add If-None-Match header if ETag provided
        if etag:
            headers["If-None-Match"] = etag
        elif policy_area in self._etag_cache:
            headers["If-None-Match"] = self._etag_cache[policy_area]

        try:
            response = self._httpx.get(
                url,
                headers=headers,
                timeout=self._timeout_s,
            )

            # Handle status codes
            if response.status_code == 200:
                # Validate response size
                content_length = len(response.content)
                if content_length > self.MAX_RESPONSE_SIZE_BYTES:
                    self._record_failure()
                    raise SignalUnavailableError(
                        f"Response size {content_length} bytes exceeds maximum "
                        f"{self.MAX_RESPONSE_SIZE_BYTES} bytes",
                        status_code=200,
                    )

                # Parse and validate with Pydantic
                data = response.json()
                signal_pack = SignalPack(**data)

                # Cache ETag
                if "ETag" in response.headers:
                    self._etag_cache[policy_area] = response.headers["ETag"]

                # Reset failure count on success
                self._failure_count = 0

                logger.info(
                    "signal_pack_fetched",
                    policy_area=policy_area,
```

```
            version=signal_pack.version,
            content_length=content_length,
        )

        return signal_pack

    elif response.status_code == 304:
        # Not Modified - cache is fresh
        logger.debug("signal_not_modified", policy_area=policy_area)
        return None

    elif response.status_code in (401, 403):
        # Authentication/Authorization error
        self._record_failure()
        raise SignalUnavailableError(
            f"Authentication failed: {response.status_code} {response.text}",
            status_code=response.status_code,
        )

    elif response.status_code == 429:
        # Rate limit - retry will handle this
        self._record_failure()
        raise SignalUnavailableError(
            "Rate limit exceeded (429 Too Many Requests)",
            status_code=429,
        )

    elif response.status_code >= 500:
        # Server error - retry will handle this
        self._record_failure()
        raise SignalUnavailableError(
            f"Server error: {response.status_code} {response.text}",
            status_code=response.status_code,
        )

    else:
        # Other error
        self._record_failure()
        raise SignalUnavailableError(
            f"Unexpected status: {response.status_code} {response.text}",
            status_code=response.status_code,
        )

except self._httpx.TimeoutException as e:
    self._record_failure()
    raise SignalUnavailableError(
        f"Request timeout after {self._timeout_s}s",
        status_code=None,
    ) from e

except self._httpx.RequestError as e:
    # Network error
    self._record_failure()
    raise SignalUnavailableError(
        f"Network error: {e}",
        status_code=None,
    ) from e

except Exception as e:
    # Unexpected error
    logger.error(
        "signal_client_fetch_failed",
        policy_area=policy_area,
        error=str(e),
        error_type=type(e).__name__,
    )
    self._record_failure()
    raise
```

```python
    def _record_failure(self) -> None:
        """Record a failure and potentially open circuit."""
        old_open = self._circuit_open

        self._failure_count += 1
        self._last_failure_time = time.time()

        if self._failure_count >= self._circuit_breaker_threshold:
            self._circuit_open = True

        # Record state change if circuit opened
        if old_open != self._circuit_open:
            self._state_changes.append({
                'timestamp': time.time(),
                'from_open': old_open,
                'to_open': self._circuit_open,
                'failures': self._failure_count,
            })

            # Trim history
            if len(self._state_changes) > self._max_history:
                self._state_changes = self._state_changes[-self._max_history:]

            logger.warning(
                "signal_client_circuit_opened",
                failure_count=self._failure_count,
                old_open=old_open,
                new_open=self._circuit_open,
            )
        else:
            # Just log the failure increment
            logger.debug(
                "signal_client_failure_recorded",
                failure_count=self._failure_count,
                threshold=self._circuit_breaker_threshold,
            )

    def get_metrics(self) -> dict[str, Any]:
        """
        Get client metrics for observability.

        Returns:
            Dict with metrics:
            - transport: Transport mode (memory or http)
            - circuit_open: Whether circuit breaker is open
            - failure_count: Current failure count
            - etag_cache_size: Number of cached ETags
            - state_change_count: Number of circuit breaker state changes
            - last_failure_time: Timestamp of last failure (or None)
        """
        return {
            "transport": self._transport,
            "circuit_open": self._circuit_open,
            "failure_count": self._failure_count,
            "etag_cache_size": len(self._etag_cache),
            "state_change_count": len(self._state_changes),
            "last_failure_time": self._last_failure_time if self._last_failure_time else
None,
        }

    def get_state_history(self) -> list[dict[str, Any]]:
        """
        Get history of circuit breaker state changes for monitoring.

        Returns:
            List of state change records with timestamps
        """
```

```python
        return list(self._state_changes)

    def register_memory_signal(self, policy_area: str, signal_pack: SignalPack) -> None:
        """
        Register signal pack in memory source (memory:// mode only).

        Args:
            policy_area: Policy area key
            signal_pack: Signal pack to register

        Raises:
            ValueError: If not in memory:// mode
        """
        if self._transport != "memory" or self._memory_source is None:
            raise ValueError("Can only register signals in memory:// mode")

        self._memory_source.register(policy_area, signal_pack)


@dataclass
class SignalUsageMetadata:
    """
    Metadata about signal usage in an execution.

    Attributes:
        version: Signal pack version used
        policy_area: Policy area of signals
        hash: Content hash of signal pack
        keys_used: List of signal keys actually used
        timestamp_utc: ISO timestamp of usage
    """

    version: str
    policy_area: str
    hash: str
    keys_used: list[str]
    timestamp_utc: str = field(
        default_factory=lambda: datetime.now(timezone.utc).isoformat()
    )

    def to_dict(self) -> dict[str, Any]:
        """Convert to dictionary for serialization."""
        return {
            "version": self.version,
            "policy_area": self.policy_area,
            "hash": self.hash,
            "keys_used": self.keys_used,
            "timestamp_utc": self.timestamp_utc,
        }


def create_default_signal_pack(policy_area: PolicyArea) -> SignalPack:
    """
    Create default signal pack for a policy area (conservative mode).

    Args:
        policy_area: Policy area

    Returns:
        SignalPack with conservative defaults
    """
    return SignalPack(
        version="0.0.0",
        policy_area=policy_area,
        patterns=[],
        indicators=[],
        regex=[],
        verbs=[],
```

```
        entities=[],
        thresholds={
            "min_confidence": 0.9,
            "min_evidence": 0.8,
        },
        ttl_s=0,  # No expiration for defaults
        source_fingerprint="default",
        metadata={"mode": "conservative_fallback"},
    )
```

===== FILE: src/saaaaaa/core/orchestrator/verification_manifest.py =====

```
"""
Verification Manifest Generation with Cryptographic Integrity

Generates verification manifests for pipeline executions with HMAC signatures
for tamper detection and comprehensive execution environment tracking.
"""

from __future__ import annotations

import hashlib
import hmac
import json
import logging
import os
import platform
import sys
from datetime import datetime
from typing import Any

logger = logging.getLogger(__name__)

# Manifest schema version
MANIFEST_VERSION = "1.0.0"


class VerificationManifest:
    """
    Builder for verification manifests with cryptographic integrity.

    Features:
    - JSON Schema validation
    - HMAC-SHA256 integrity signatures
    - Execution environment tracking
    - Determinism metadata
    - Phase and artifact tracking
    """

    def __init__(self, hmac_secret: str | None = None) -> None:
        """
        Initialize manifest builder.

        Args:
            hmac_secret: Secret key for HMAC generation. If None, uses
                        environment variable VERIFICATION_HMAC_SECRET.
                        If not set, generates warning (integrity disabled).
        """
        self.hmac_secret = hmac_secret or os.getenv("VERIFICATION_HMAC_SECRET")
        if not self.hmac_secret:
            logger.warning(
                "No HMAC secret provided. Integrity verification disabled. "
                "Set VERIFICATION_HMAC_SECRET environment variable."
            )

        self.manifest_data: dict[str, Any] = {
            "version": MANIFEST_VERSION,
            "timestamp": datetime.utcnow().isoformat() + "Z",
            "success": False,  # Default to false, set explicitly
```

```
    }

def set_success(self, success: bool):
    """Set overall pipeline success flag."""
    self.manifest_data["success"] = success
    return self

def set_pipeline_hash(self, pipeline_hash: str):
    """Set SHA256 hash of pipeline execution."""
    self.manifest_data["pipeline_hash"] = pipeline_hash
    return self

def set_environment(self):
    """
    Capture execution environment information.

    Automatically captures:
    - Python version
    - Platform (OS)
    - CPU count
    - Available memory (if psutil available)
    - UTC timestamp
    """
    env_data = {
        "python_version": sys.version,
        "platform": platform.platform(),
        "cpu_count": os.cpu_count() or 1,
        "timestamp_utc": datetime.utcnow().isoformat() + "Z",
    }

    # Try to get memory info
    try:
        import psutil
        mem = psutil.virtual_memory()
        env_data["memory_gb"] = round(mem.total / (1024**3), 2)
    except ImportError:
        logger.debug("psutil not available, skipping memory info")
    except Exception as e:
        logger.debug(f"Failed to get memory info: {e}")

    self.manifest_data["environment"] = env_data
    return self

def add_environment_info(self, environment: dict[str, Any] | None = None):
    """
    Merge extra environment attributes into the manifest.

    Args:
        environment: Optional mapping of additional environment data.
    """
    if environment:
        current = self.manifest_data.get("environment", {})
        current.update(environment)
        self.manifest_data["environment"] = current
    elif "environment" not in self.manifest_data:
        self.set_environment()
    return self

def set_determinism(
    self,
    seed_version: str,
    base_seed: int | None = None,
    policy_unit_id: str | None = None,
    correlation_id: str | None = None,
    seeds_by_component: dict[str, int] | None = None
):
    """
    Set determinism tracking information.
```

```
    Args:
        seed_version: Seed derivation algorithm version
        base_seed: Base seed used
        policy_unit_id: Policy unit identifier
        correlation_id: Execution correlation ID
        seeds_by_component: Dict mapping component names to seeds
    """
    determinism_data = {
        "seed_version": seed_version
    }

    if base_seed is not None:
        determinism_data["base_seed"] = base_seed
    if policy_unit_id:
        determinism_data["policy_unit_id"] = policy_unit_id
    if correlation_id:
        determinism_data["correlation_id"] = correlation_id
    if seeds_by_component:
        determinism_data["seeds_by_component"] = seeds_by_component

    self.manifest_data["determinism"] = determinism_data
    return self

def set_determinism_info(self, determinism_info: dict[str, Any]):
    """Alias for setting determinism metadata directly."""
    if determinism_info:
        self.manifest_data["determinism"] = determinism_info
    return self

def set_calibrations(
    self,
    version: str,
    calibration_hash: str,
    methods_calibrated: int,
    methods_missing: list[str]
):
    """
    Set calibration information.

    Args:
        version: Calibration registry version
        calibration_hash: SHA256 hash of calibration data
        methods_calibrated: Number of calibrated methods
        methods_missing: List of methods without calibration
    """
    self.manifest_data["calibrations"] = {
        "version": version,
        "hash": calibration_hash,
        "methods_calibrated": methods_calibrated,
        "methods_missing": methods_missing
    }
    return self

def set_calibration_info(self, calibration_info: dict[str, Any]):
    """Set calibration metadata using a snapshot dictionary."""
    if calibration_info:
        self.manifest_data["calibration"] = calibration_info
    return self

def set_ingestion(
    self,
    method: str,
    chunk_count: int,
    text_length: int,
    sentence_count: int,
    chunk_strategy: str | None = None,
    chunk_overlap: int | None = None
```

```python
):
    """
    Set ingestion information.

    Args:
        method: Ingestion method ("SPC" or "CPP")
        chunk_count: Number of chunks
        text_length: Total text length
        sentence_count: Number of sentences
        chunk_strategy: Chunking strategy used
        chunk_overlap: Chunk overlap in characters
    """
    ingestion_data = {
        "method": method,
        "chunk_count": chunk_count,
        "text_length": text_length,
        "sentence_count": sentence_count
    }

    if chunk_strategy:
        ingestion_data["chunk_strategy"] = chunk_strategy
    if chunk_overlap is not None:
        ingestion_data["chunk_overlap"] = chunk_overlap

    self.manifest_data["ingestion"] = ingestion_data
    return self

def set_parametrization(self, parametrization: dict[str, Any]):
    """Record executor/config parameterization data."""
    if parametrization:
        self.manifest_data["parametrization"] = parametrization
    return self

def add_phase(
    self,
    phase_id: int,
    phase_name: str,
    success: bool,
    duration_ms: float | None = None,
    items_processed: int | None = None,
    error: str | None = None
):
    """
    Add phase execution information.

    Args:
        phase_id: Phase numeric identifier
        phase_name: Phase human-readable name
        success: Phase execution success
        duration_ms: Duration in milliseconds
        items_processed: Number of items processed
        error: Error message if failed
    """
    if "phases" not in self.manifest_data:
        self.manifest_data["phases"] = []

    phase_data = {
        "phase_id": phase_id,
        "phase_name": phase_name,
        "success": success
    }

    if duration_ms is not None:
        phase_data["duration_ms"] = duration_ms
    if items_processed is not None:
        phase_data["items_processed"] = items_processed
    if error:
        phase_data["error"] = error
```

```
    container = self.manifest_data.get("phases")
    if isinstance(container, dict):
        entries = container.setdefault("entries", [])
        entries.append(phase_data)
    else:
        container.append(phase_data)
    return self

def add_artifact(
    self,
    artifact_id: str,
    path: str,
    artifact_hash: str,
    size_bytes: int | None = None
):
    """
    Add artifact information.

    Args:
        artifact_id: Artifact identifier
        path: Artifact file path
        artifact_hash: SHA256 hash of artifact
        size_bytes: Artifact size in bytes
    """
    if "artifacts" not in self.manifest_data:
        self.manifest_data["artifacts"] = {}

    artifact_data = {
        "path": path,
        "hash": artifact_hash
    }

    if size_bytes is not None:
        artifact_data["size_bytes"] = size_bytes

    self.manifest_data["artifacts"][artifact_id] = artifact_data
    return self

def _compute_hmac(self, content: str) -> str:
    """
    Compute HMAC-SHA256 of manifest content.

    Args:
        content: JSON string of manifest (without HMAC field)

    Returns:
        Hex-encoded HMAC signature
    """
    if not self.hmac_secret:
        return "00" * 32  # Placeholder if no secret

    signature = hmac.new(
        self.hmac_secret.encode("utf-8"),
        content.encode("utf-8"),
        hashlib.sha256
    )
    return signature.hexdigest()

def build(self) -> dict[str, Any]:
    """
    Build final manifest with HMAC signature.

    Returns:
        Complete manifest dictionary with integrity_hmac
    """
    # Create canonical JSON (without HMAC)
    canonical = json.dumps(
```

```python
            self.manifest_data,
            sort_keys=True,
            indent=None,
            separators=(',', ':')
        )

        # Compute HMAC
        hmac_signature = self._compute_hmac(canonical)

        # Add HMAC to manifest
        final_manifest = dict(self.manifest_data)
        final_manifest["integrity_hmac"] = hmac_signature

        return final_manifest

    def build_json(self, indent: int = 2) -> str:
        """
        Build manifest as JSON string.

        Args:
            indent: JSON indentation level

        Returns:
            Pretty-printed JSON string
        """
        manifest = self.build()
        return json.dumps(manifest, indent=indent)

    def write(self, filepath: str) -> None:
        """
        Write manifest to file.

        Args:
            filepath: Path to write manifest JSON
        """
        manifest_json = self.build_json()

        with open(filepath, 'w', encoding='utf-8') as f:
            f.write(manifest_json)

        logger.info(f"Verification manifest written to: {filepath}")


def verify_manifest_integrity(
    manifest: dict[str, Any],
    hmac_secret: str
) -> bool:
    """
    Verify HMAC integrity of a manifest.

    Args:
        manifest: Manifest dictionary (with integrity_hmac)
        hmac_secret: HMAC secret key

    Returns:
        True if HMAC is valid, False otherwise
    """
    if "integrity_hmac" not in manifest:
        logger.error("Manifest missing integrity_hmac field")
        return False

    # Extract HMAC
    provided_hmac = manifest["integrity_hmac"]

    # Rebuild manifest without HMAC
    manifest_without_hmac = {k: v for k, v in manifest.items() if k != "integrity_hmac"}

    # Compute canonical JSON
```

```python
    canonical = json.dumps(
        manifest_without_hmac,
        sort_keys=True,
        indent=None,
        separators=(',', ':')
    )

    # Compute expected HMAC
    expected_hmac = hmac.new(
        hmac_secret.encode("utf-8"),
        canonical.encode("utf-8"),
        hashlib.sha256
    ).hexdigest()

    # Constant-time comparison
    is_valid = hmac.compare_digest(provided_hmac, expected_hmac)

    if not is_valid:
        logger.error("HMAC verification failed - manifest may be tampered")

    return is_valid
```

===== FILE: src/saaaaaa/core/orchestrator/versions.py =====

```python
"""
Version Tracking for Contract Enforcement

Centralized version management for all contract enforcement components.
Enables compatibility checking and rollback safety.
"""

# Pipeline version
PIPELINE_VERSION = "2.0.0"

# Calibration version (from calibration_registry.py)
CALIBRATION_VERSION = "2.0.0"  # Should match calibration_registry.CALIBRATION_VERSION

# Signal version
SIGNAL_VERSION = "1.0.0"

# Advanced module version
ADVANCED_MODULE_VERSION = "1.0.0"

# Seed registry version
SEED_VERSION = "sha256_v1"  # Should match seed_registry.SEED_VERSION

# Verification manifest version
MANIFEST_VERSION = "1.0.0"  # Should match verification_manifest.MANIFEST_VERSION

# Minimum supported versions for backward compatibility
MIN_CALIBRATION_VERSION = "2.0.0"
MIN_SIGNAL_VERSION = "1.0.0"
MIN_PIPELINE_VERSION = "2.0.0"


def check_version_compatibility(component: str, version: str, min_version: str) -> bool:
    """
    Check if a version meets minimum requirements.

    Args:
        component: Component name (for error messages)
        version: Current version string (e.g., "2.1.0")
        min_version: Minimum required version (e.g., "2.0.0")

    Returns:
        True if version >= min_version

    Raises:
        ValueError: If version is incompatible
```

```
    """
    try:
        v_parts = [int(x) for x in version.split(".")]
        min_parts = [int(x) for x in min_version.split(".")]

        # Pad to same length
        while len(v_parts) < len(min_parts):
            v_parts.append(0)
        while len(min_parts) < len(v_parts):
            min_parts.append(0)

        # Compare tuple
        if tuple(v_parts) < tuple(min_parts):
            raise ValueError(
                f"{component} version {version} is below minimum required {min_version}. "
                "Please upgrade or regenerate calibration data."
            )

        return True
    except (ValueError, AttributeError) as e:
        if "below minimum" in str(e):
            raise
        raise ValueError(
            f"Invalid version format for {component}: {version}. "
            "Expected semantic version like '1.0.0'"
        ) from e


def get_all_versions() -> dict[str, str]:
    """
    Get all component versions for manifest inclusion.

    Returns:
        Dictionary mapping component names to version strings
    """
    return {
        "pipeline": PIPELINE_VERSION,
        "calibration": CALIBRATION_VERSION,
        "signal": SIGNAL_VERSION,
        "advanced_module": ADVANCED_MODULE_VERSION,
        "seed": SEED_VERSION,
        "manifest": MANIFEST_VERSION,
    }

===== FILE: src/saaaaaa/core/phases/phase0_input_validation.py =====
"""
Phase 0: Input Validation - Constitutional Implementation
============================================================

This module implements Phase 0 of the canonical pipeline:
    Raw input → Validated CanonicalInput

Responsibilities:
-----------------
1. Validate PDF exists and is readable
2. Compute SHA256 hash of PDF (deterministic fingerprint)
3. Extract PDF metadata (page count, size)
4. Validate questionnaire exists
5. Compute SHA256 hash of questionnaire
6. Package validated inputs into CanonicalInput

Input Contract:
---------------
Phase0Input:
    - pdf_path: Path (must exist)
    - run_id: str (unique execution identifier)
    - questionnaire_path: Path | None (optional, defaults to canonical)
```

```
Output Contract:
----------------
CanonicalInput:
    - document_id: str (derived from PDF stem or explicit)
    - run_id: str (preserved from input)
    - pdf_path: Path (validated)
    - pdf_sha256: str (computed hash)
    - pdf_size_bytes: int (file size)
    - pdf_page_count: int (extracted from PDF)
    - questionnaire_path: Path (validated)
    - questionnaire_sha256: str (computed hash)
    - created_at: datetime (UTC timestamp)
    - phase0_version: str (schema version)
    - validation_passed: bool (must be True for output)
    - validation_errors: list[str] (empty if passed)
    - validation_warnings: list[str] (may contain warnings)

Invariants:
-----------
1. validation_passed == True
2. pdf_page_count > 0
3. pdf_size_bytes > 0
4. pdf_sha256 is 64-char hex string
5. questionnaire_sha256 is 64-char hex string

Author: F.A.R.F.A.N Architecture Team
Date: 2025-01-19
"""

from __future__ import annotations

import hashlib
from dataclasses import dataclass, field
from datetime import datetime, timezone
from pathlib import Path
from typing import Any

from pydantic import BaseModel, Field, field_validator

from saaaaaa.core.phases.phase_protocol import (
    ContractValidationResult,
    PhaseContract,
)

# Schema version for Phase 0
PHASE0_VERSION = "1.0.0"


# ============================================================================
# INPUT CONTRACT
# ============================================================================


@dataclass
class Phase0Input:
    """
    Input contract for Phase 0.

    This is the raw, unvalidated input to the pipeline.
    """

    pdf_path: Path
    run_id: str
    questionnaire_path: Path | None = None


class Phase0InputValidator(BaseModel):
    """Pydantic validator for Phase0Input."""
```

```python
    pdf_path: str = Field(description="Path to input PDF")
    run_id: str = Field(min_length=1, description="Unique run identifier")
    questionnaire_path: str | None = Field(
        default=None, description="Optional questionnaire path"
    )

    @field_validator("pdf_path")
    @classmethod
    def validate_pdf_path(cls, v: str) -> str:
        """Validate PDF path format."""
        if not v or not v.strip():
            raise ValueError("pdf_path cannot be empty")
        return v

    @field_validator("run_id")
    @classmethod
    def validate_run_id(cls, v: str) -> str:
        """Validate run_id format."""
        if not v or not v.strip():
            raise ValueError("run_id cannot be empty")
        # Ensure run_id is filesystem-safe
        if any(char in v for char in ['/', '\\', ':', '*', '?', '"', '<', '>', '|']):
            raise ValueError(
                "run_id contains invalid characters (must be filesystem-safe)"
            )
        return v


# ============================================================================
# OUTPUT CONTRACT
# ============================================================================


@dataclass
class CanonicalInput:
    """
    Output contract for Phase 0.

    This represents a validated, canonical input ready for Phase 1.
    All fields are required and validated.
    """

    # Identity
    document_id: str
    run_id: str

    # Input artifacts (immutable, validated)
    pdf_path: Path
    pdf_sha256: str
    pdf_size_bytes: int
    pdf_page_count: int

    # Questionnaire (required for SIN_CARRETA compliance)
    questionnaire_path: Path
    questionnaire_sha256: str

    # Metadata
    created_at: datetime
    phase0_version: str

    # Validation results
    validation_passed: bool
    validation_errors: list[str] = field(default_factory=list)
    validation_warnings: list[str] = field(default_factory=list)


class CanonicalInputValidator(BaseModel):
```

```python
    """Pydantic validator for CanonicalInput."""

    document_id: str = Field(min_length=1)
    run_id: str = Field(min_length=1)
    pdf_path: str
    pdf_sha256: str = Field(min_length=64, max_length=64)
    pdf_size_bytes: int = Field(gt=0)
    pdf_page_count: int = Field(gt=0)
    questionnaire_path: str
    questionnaire_sha256: str = Field(min_length=64, max_length=64)
    created_at: str
    phase0_version: str
    validation_passed: bool
    validation_errors: list[str] = Field(default_factory=list)
    validation_warnings: list[str] = Field(default_factory=list)

    @field_validator("validation_passed")
    @classmethod
    def validate_passed(cls, v: bool, info) -> bool:
        """Ensure validation_passed is True and consistent with errors."""
        if not v:
            raise ValueError(
                "validation_passed must be True for valid CanonicalInput"
            )
        errors = info.data.get("validation_errors", [])
        if errors:
            raise ValueError(
                f"validation_passed is True but validation_errors is not empty: {errors}"
            )
        return v

    @field_validator("pdf_sha256", "questionnaire_sha256")
    @classmethod
    def validate_sha256(cls, v: str) -> str:
        """Validate SHA256 hash format."""
        if len(v) != 64:
            raise ValueError(f"SHA256 hash must be 64 characters, got {len(v)}")
        if not all(c in "0123456789abcdef" for c in v.lower()):
            raise ValueError("SHA256 hash must be hexadecimal")
        return v.lower()


# ============================================================================
# PHASE 0 CONTRACT IMPLEMENTATION
# ============================================================================


class Phase0ValidationContract(PhaseContract[Phase0Input, CanonicalInput]):
    """
    Phase 0: Input Validation Contract.

    This class enforces the constitutional constraint that Phase 0:
    1. Accepts ONLY Phase0Input
    2. Produces ONLY CanonicalInput
    3. Validates all invariants
    4. Logs all operations
    """

    def __init__(self):
        """Initialize Phase 0 contract with invariants."""
        super().__init__(phase_name="phase0_input_validation")

        # Register invariants
        self.add_invariant(
            name="validation_passed",
            description="Output must have validation_passed=True",
            check=lambda data: data.validation_passed is True,
            error_message="validation_passed must be True",
```

```
        )

        self.add_invariant(
            name="pdf_page_count_positive",
            description="PDF must have at least 1 page",
            check=lambda data: data.pdf_page_count > 0,
            error_message="pdf_page_count must be > 0",
        )

        self.add_invariant(
            name="pdf_size_positive",
            description="PDF size must be > 0 bytes",
            check=lambda data: data.pdf_size_bytes > 0,
            error_message="pdf_size_bytes must be > 0",
        )

        self.add_invariant(
            name="sha256_format",
            description="SHA256 hashes must be valid",
            check=lambda data: (
                len(data.pdf_sha256) == 64
                and len(data.questionnaire_sha256) == 64
                and all(c in "0123456789abcdef" for c in data.pdf_sha256.lower())
                and all(c in "0123456789abcdef" for c in
data.questionnaire_sha256.lower())
            ),
            error_message="SHA256 hashes must be 64-char hexadecimal",
        )

        self.add_invariant(
            name="no_validation_errors",
            description="validation_errors must be empty",
            check=lambda data: len(data.validation_errors) == 0,
            error_message="validation_errors must be empty for valid output",
        )

    def validate_input(self, input_data: Any) -> ContractValidationResult:
        """
        Validate Phase0Input contract.

        Args:
            input_data: Input to validate

        Returns:
            ContractValidationResult
        """
        errors = []
        warnings = []

        # Type check
        if not isinstance(input_data, Phase0Input):
            errors.append(
                f"Expected Phase0Input, got {type(input_data).__name__}"
            )
            return ContractValidationResult(
                passed=False,
                contract_type="input",
                phase_name=self.phase_name,
                errors=errors,
            )

        # Validate using Pydantic
        try:
            Phase0InputValidator(
                pdf_path=str(input_data.pdf_path),
                run_id=input_data.run_id,
                questionnaire_path=(
                    str(input_data.questionnaire_path)
```

```python
                if input_data.questionnaire_path
                else None
            ),
        )
    except Exception as e:
        errors.append(f"Pydantic validation failed: {e}")

    return ContractValidationResult(
        passed=len(errors) == 0,
        contract_type="input",
        phase_name=self.phase_name,
        errors=errors,
        warnings=warnings,
    )

def validate_output(self, output_data: Any) -> ContractValidationResult:
    """
    Validate CanonicalInput contract.

    Args:
        output_data: Output to validate

    Returns:
        ContractValidationResult
    """
    errors = []
    warnings = []

    # Type check
    if not isinstance(output_data, CanonicalInput):
        errors.append(
            f"Expected CanonicalInput, got {type(output_data).__name__}"
        )
        return ContractValidationResult(
            passed=False,
            contract_type="output",
            phase_name=self.phase_name,
            errors=errors,
        )

    # Validate using Pydantic
    try:
        CanonicalInputValidator(
            document_id=output_data.document_id,
            run_id=output_data.run_id,
            pdf_path=str(output_data.pdf_path),
            pdf_sha256=output_data.pdf_sha256,
            pdf_size_bytes=output_data.pdf_size_bytes,
            pdf_page_count=output_data.pdf_page_count,
            questionnaire_path=str(output_data.questionnaire_path),
            questionnaire_sha256=output_data.questionnaire_sha256,
            created_at=output_data.created_at.isoformat(),
            phase0_version=output_data.phase0_version,
            validation_passed=output_data.validation_passed,
            validation_errors=output_data.validation_errors,
            validation_warnings=output_data.validation_warnings,
        )
    except Exception as e:
        errors.append(f"Pydantic validation failed: {e}")

    return ContractValidationResult(
        passed=len(errors) == 0,
        contract_type="output",
        phase_name=self.phase_name,
        errors=errors,
        warnings=warnings,
    )
```

```python
async def execute(self, input_data: Phase0Input) -> CanonicalInput:
    """
    Execute Phase 0: Input Validation.

    Args:
        input_data: Phase0Input with raw paths

    Returns:
        CanonicalInput with validated data

    Raises:
        FileNotFoundError: If PDF or questionnaire doesn't exist
        ValueError: If validation fails
    """
    errors = []
    warnings = []

    # 1. Resolve questionnaire path
    questionnaire_path = input_data.questionnaire_path
    if questionnaire_path is None:
        from saaaaaa.config.paths import QUESTIONNAIRE_FILE

        questionnaire_path = QUESTIONNAIRE_FILE
        warnings.append(
            f"questionnaire_path not provided, using default: {questionnaire_path}"
        )

    # 2. Validate PDF exists
    if not input_data.pdf_path.exists():
        errors.append(f"PDF not found: {input_data.pdf_path}")
    if not input_data.pdf_path.is_file():
        errors.append(f"PDF path is not a file: {input_data.pdf_path}")

    # 3. Validate questionnaire exists
    if not questionnaire_path.exists():
        errors.append(f"Questionnaire not found: {questionnaire_path}")
    if not questionnaire_path.is_file():
        errors.append(f"Questionnaire path is not a file: {questionnaire_path}")

    # If basic validation failed, abort
    if errors:
        raise FileNotFoundError(f"Input validation failed: {errors}")

    # 4. Compute PDF hash and metadata
    pdf_sha256 = self._compute_sha256(input_data.pdf_path)
    pdf_size_bytes = input_data.pdf_path.stat().st_size
    pdf_page_count = self._get_pdf_page_count(input_data.pdf_path)

    # 5. Compute questionnaire hash
    questionnaire_sha256 = self._compute_sha256(questionnaire_path)

    # 6. Determine document_id
    document_id = input_data.pdf_path.stem

    # 7. Create CanonicalInput
    canonical_input = CanonicalInput(
        document_id=document_id,
        run_id=input_data.run_id,
        pdf_path=input_data.pdf_path,
        pdf_sha256=pdf_sha256,
        pdf_size_bytes=pdf_size_bytes,
        pdf_page_count=pdf_page_count,
        questionnaire_path=questionnaire_path,
        questionnaire_sha256=questionnaire_sha256,
        created_at=datetime.now(timezone.utc),
        phase0_version=PHASE0_VERSION,
        validation_passed=len(errors) == 0,
        validation_errors=errors,
```

```
            validation_warnings=warnings,
        )

        return canonical_input

    @staticmethod
    def _compute_sha256(file_path: Path) -> str:
        """
        Compute SHA256 hash of a file.

        Args:
            file_path: Path to file

        Returns:
            Hex-encoded SHA256 hash (lowercase)
        """
        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().lower()

    @staticmethod
    def _get_pdf_page_count(pdf_path: Path) -> int:
        """
        Extract page count from PDF.

        Args:
            pdf_path: Path to PDF file

        Returns:
            Number of pages

        Raises:
            ImportError: If PyMuPDF is not available
            RuntimeError: If PDF cannot be opened
        """
        try:
            import fitz  # PyMuPDF

            doc = fitz.open(pdf_path)
            page_count = len(doc)
            doc.close()
            return page_count
        except ImportError:
            raise ImportError(
                "PyMuPDF (fitz) required for PDF page count extraction. "
                "Install with: pip install PyMuPDF"
            )
        except Exception as e:
            raise RuntimeError(f"Failed to open PDF {pdf_path}: {e}")


__all__ = [
    "Phase0Input",
    "CanonicalInput",
    "Phase0ValidationContract",
    "PHASE0_VERSION",
]
```

===== FILE: src/saaaaaa/core/phases/phase1_spc_ingestion.py =====

```
"""
Phase 1: SPC Ingestion - Constitutional Implementation
=========================================================

This module implements Phase 1 of the canonical pipeline:
    CanonicalInput → CanonPolicyPackage
```

```
Responsibilities:
-----------------
1. Load and validate document from CanonicalInput
2. Execute 15 subfases of Strategic Chunking System
3. Generate exactly 60 chunks structured as 10 PA × 6 DIM
4. Validate quality metrics and integrity
5. Package results into CanonPolicyPackage

Input Contract:
---------------
CanonicalInput (from Phase 0):
    - document_id, pdf_path, pdf_sha256
    - questionnaire_path
    - All validation passed

Output Contract:
----------------
CanonPolicyPackage:
    - schema_version: str ("SPC-2025.1")
    - document_id: str (preserved from input)
    - chunk_graph: ChunkGraph (60 chunks, PA×DIM)
    - policy_manifest: PolicyManifest
    - quality_metrics: QualityMetrics
    - integrity_index: IntegrityIndex
    - metadata: dict

15 Subfases (Internal to Phase 1):
-----------------------------------
Subfase 0:  Language detection & model selection
Subfase 1:  Advanced preprocessing
Subfase 2:  Structural analysis & hierarchy extraction
Subfase 3:  Topic modeling & global KG construction
Subfase 4:  Structured (PA×DIM) segmentation → 60 chunks
Subfase 5:  Complete causal chain extraction
Subfase 6:  Causal integration
Subfase 7:  Deep argumentative analysis
Subfase 8:  Temporal and sequential analysis
Subfase 9:  Discourse and rhetorical analysis
Subfase 10: Multi-scale strategic integration
Subfase 11: Smart Policy Chunk generation
Subfase 12: Inter-chunk relationship enrichment
Subfase 13: Strategic integrity validation
Subfase 14: Intelligent deduplication
Subfase 15: Strategic importance ranking

Invariants:
-----------
1. chunk_count == 60 (10 PA × 6 DIM)
2. All chunks have policy_area_id (PA01-PA10)
3. All chunks have dimension_id (DIM01-DIM06)
4. quality_metrics.provenance_completeness >= 0.8
5. quality_metrics.structural_consistency >= 0.85
6. All chunks pass integrity validation

Author: F.A.R.F.A.N Architecture Team
Date: 2025-01-19
"""

from __future__ import annotations

import logging
from dataclasses import dataclass, field
from pathlib import Path
from typing import Any

from pydantic import BaseModel, Field, field_validator

from saaaaaa.core.phases.phase_protocol import (
```

```python
    ContractValidationResult,
    PhaseContract,
)
from saaaaaa.core.phases.phase0_input_validation import CanonicalInput
from saaaaaa.processing.cpp_ingestion.models import CanonPolicyPackage

logger = logging.getLogger(__name__)

# Schema version for Phase 1
PHASE1_VERSION = "SPC-2025.1"

# Expected chunk count (10 PA × 6 DIM)
EXPECTED_CHUNK_COUNT = 60

# Policy Areas (PA01-PA10)
POLICY_AREAS = [
    "PA01",  # Mujeres y equidad de género
    "PA02",  # Paz, seguridad y convivencia
    "PA03",  # Ambiente y cambio climático
    "PA04",  # Derechos económicos, sociales y culturales
    "PA05",  # Víctimas y construcción de paz
    "PA06",  # Niñez, adolescencia y juventud
    "PA07",  # Tierras y territorios
    "PA08",  # Líderes y defensores de DDHH
    "PA09",  # Privadas de libertad
    "PA10",  # Migración transfronteriza
]

# Dimensions (DIM01-DIM06)
DIMENSIONS = [
    "DIM01",  # Diagnóstico y recursos
    "DIM02",  # Actividades e intervenciones
    "DIM03",  # Productos (outputs)
    "DIM04",  # Resultados (outcomes)
    "DIM05",  # Impactos de largo plazo
    "DIM06",  # Causalidad y teoría de cambio
]


# ============================================================================
# SUBFASE TRACKING
# ============================================================================


@dataclass
class SubfaseMetadata:
    """Metadata for a single subfase execution."""

    subfase_number: int
    subfase_name: str
    started_at: str
    finished_at: str | None = None
    duration_ms: float | None = None
    success: bool = False
    error: str | None = None


# ============================================================================
# OUTPUT CONTRACT VALIDATOR
# ============================================================================


class CanonPolicyPackageValidator(BaseModel):
    """Pydantic validator for CanonPolicyPackage."""

    schema_version: str = Field(description="Must be SPC-2025.1")
    document_id: str = Field(min_length=1)
    chunk_count: int = Field(ge=1, description="Number of chunks in chunk_graph")
```

```python
    has_policy_manifest: bool
    has_quality_metrics: bool
    has_integrity_index: bool
    provenance_completeness: float = Field(ge=0.0, le=1.0)
    structural_consistency: float = Field(ge=0.0, le=1.0)

    @field_validator("schema_version")
    @classmethod
    def validate_schema_version(cls, v: str) -> str:
        """Ensure schema version is correct."""
        if v != PHASE1_VERSION:
            raise ValueError(
                f"schema_version must be '{PHASE1_VERSION}', got '{v}'"
            )
        return v

    @field_validator("chunk_count")
    @classmethod
    def validate_chunk_count(cls, v: int) -> int:
        """Validate chunk count."""
        if v != EXPECTED_CHUNK_COUNT:
            raise ValueError(
                f"Expected {EXPECTED_CHUNK_COUNT} chunks (10 PA × 6 DIM), got {v}"
            )
        return v

    @field_validator("provenance_completeness")
    @classmethod
    def validate_provenance(cls, v: float) -> float:
        """Ensure provenance completeness meets threshold."""
        if v < 0.8:
            raise ValueError(
                f"provenance_completeness must be >= 0.8, got {v:.2f}"
            )
        return v

    @field_validator("structural_consistency")
    @classmethod
    def validate_structural(cls, v: float) -> float:
        """Ensure structural consistency meets threshold."""
        if v < 0.85:
            raise ValueError(
                f"structural_consistency must be >= 0.85, got {v:.2f}"
            )
        return v


# ============================================================================
# PHASE 1 CONTRACT IMPLEMENTATION
# ============================================================================


class Phase1SPCIngestionContract(PhaseContract[CanonicalInput, CanonPolicyPackage]):
    """
    Phase 1: SPC Ingestion Contract.

    This class enforces the constitutional constraint that Phase 1:
    1. Accepts ONLY CanonicalInput (from Phase 0)
    2. Produces ONLY CanonPolicyPackage
    3. Executes all 15 subfases in order
    4. Generates exactly 60 chunks (10 PA × 6 DIM)
    5. Validates all quality metrics
    """

    def __init__(self):
        """Initialize Phase 1 contract with invariants."""
        super().__init__(phase_name="phase1_spc_ingestion")
```

```
        # Track subfases
        self.subfases: list[SubfaseMetadata] = []

        # Register invariants
        self.add_invariant(
            name="chunk_count_60",
            description="Must have exactly 60 chunks (10 PA × 6 DIM)",
            check=lambda data: len(data.chunk_graph.chunks) == EXPECTED_CHUNK_COUNT,
            error_message=f"chunk_count must be {EXPECTED_CHUNK_COUNT}",
        )

        self.add_invariant(
            name="all_chunks_have_pa",
            description="All chunks must have policy_area_id (PA01-PA10)",
            check=lambda data: all(
                chunk.policy_area_id in POLICY_AREAS
                for chunk in data.chunk_graph.chunks.values()
            ),
            error_message="All chunks must have valid policy_area_id",
        )

        self.add_invariant(
            name="all_chunks_have_dim",
            description="All chunks must have dimension_id (DIM01-DIM06)",
            check=lambda data: all(
                chunk.dimension_id in DIMENSIONS
                for chunk in data.chunk_graph.chunks.values()
            ),
            error_message="All chunks must have valid dimension_id",
        )

        self.add_invariant(
            name="provenance_threshold",
            description="Provenance completeness >= 0.8",
            check=lambda data: (
                data.quality_metrics is not None
                and data.quality_metrics.provenance_completeness >= 0.8
            ),
            error_message="provenance_completeness must be >= 0.8",
        )

        self.add_invariant(
            name="structural_threshold",
            description="Structural consistency >= 0.85",
            check=lambda data: (
                data.quality_metrics is not None
                and data.quality_metrics.structural_consistency >= 0.85
            ),
            error_message="structural_consistency must be >= 0.85",
        )

    def validate_input(self, input_data: Any) -> ContractValidationResult:
        """
        Validate CanonicalInput contract.

        Args:
            input_data: Input to validate

        Returns:
            ContractValidationResult
        """
        errors = []
        warnings = []

        # Type check
        if not isinstance(input_data, CanonicalInput):
            errors.append(
                f"Expected CanonicalInput, got {type(input_data).__name__}"
```

```
            )
            return ContractValidationResult(
                passed=False,
                contract_type="input",
                phase_name=self.phase_name,
                errors=errors,
            )

        # Validate fields
        if not input_data.validation_passed:
            errors.append(
                f"CanonicalInput has validation_passed=False:
{input_data.validation_errors}"
            )

        if not input_data.pdf_path.exists():
            errors.append(f"PDF path does not exist: {input_data.pdf_path}")

        if input_data.pdf_page_count <= 0:
            errors.append(
                f"Invalid pdf_page_count: {input_data.pdf_page_count}"
            )

        return ContractValidationResult(
            passed=len(errors) == 0,
            contract_type="input",
            phase_name=self.phase_name,
            errors=errors,
            warnings=warnings,
        )

    def validate_output(self, output_data: Any) -> ContractValidationResult:
        """
        Validate CanonPolicyPackage contract.

        Args:
            output_data: Output to validate

        Returns:
            ContractValidationResult
        """
        errors = []
        warnings = []

        # Type check
        if not isinstance(output_data, CanonPolicyPackage):
            errors.append(
                f"Expected CanonPolicyPackage, got {type(output_data).__name__}"
            )
            return ContractValidationResult(
                passed=False,
                contract_type="output",
                phase_name=self.phase_name,
                errors=errors,
            )

        # Validate using Pydantic
        try:
            CanonPolicyPackageValidator(
                schema_version=output_data.schema_version,
                document_id=output_data.metadata.get("document_id", ""),
                chunk_count=len(output_data.chunk_graph.chunks),
                has_policy_manifest=output_data.policy_manifest is not None,
                has_quality_metrics=output_data.quality_metrics is not None,
                has_integrity_index=output_data.integrity_index is not None,
                provenance_completeness=(
                    output_data.quality_metrics.provenance_completeness
                    if output_data.quality_metrics
```

```
            else 0.0
        ),
        structural_consistency=(
            output_data.quality_metrics.structural_consistency
            if output_data.quality_metrics
            else 0.0
        ),
    )
except Exception as e:
    errors.append(f"Pydantic validation failed: {e}")

return ContractValidationResult(
    passed=len(errors) == 0,
    contract_type="output",
    phase_name=self.phase_name,
    errors=errors,
    warnings=warnings,
)

async def execute(self, input_data: CanonicalInput) -> CanonPolicyPackage:
    """
    Execute Phase 1: SPC Ingestion with 15 subfases.

    Args:
        input_data: CanonicalInput from Phase 0

    Returns:
        CanonPolicyPackage with 60 chunks

    Raises:
        ImportError: If SPC pipeline not available
        ValueError: If ingestion fails
    """
    logger.info(f"Starting Phase 1: SPC Ingestion for {input_data.document_id}")

    # Import SPC pipeline
    try:
        from saaaaaa.processing.spc_ingestion import CPPIngestionPipeline
    except ImportError as e:
        raise ImportError(
            "SPC ingestion pipeline not available. "
            "Ensure saaaaaa.processing.spc_ingestion is installed."
        ) from e

    # Initialize pipeline with questionnaire
    pipeline = CPPIngestionPipeline(
        questionnaire_path=input_data.questionnaire_path,
        enable_runtime_validation=True,
    )

    # The pipeline.process() method internally executes all 15 subfases:
    # Subfase 0:  Language detection (in generate_smart_chunks)
    # Subfase 1:  Advanced preprocessing
    # Subfase 2:  Structural analysis
    # Subfase 3:  Topic modeling & KG
    # Subfase 4:  PA×DIM segmentation (60 chunks)
    # Subfase 5:  Causal chain extraction
    # Subfase 6:  Causal integration
    # Subfase 7:  Argumentative analysis
    # Subfase 8:  Temporal analysis
    # Subfase 9:  Discourse analysis
    # Subfase 10: Strategic integration
    # Subfase 11: Chunk generation
    # Subfase 12: Inter-chunk enrichment
    # Subfase 13: Integrity validation
    # Subfase 14: Deduplication
    # Subfase 15: Strategic ranking
```

```
        logger.info("Executing 15 subfases of Strategic Chunking System...")

        cpp = await pipeline.process(
            document_path=input_data.pdf_path,
            document_id=input_data.document_id,
            title=input_data.pdf_path.name,
            max_chunks=EXPECTED_CHUNK_COUNT,
        )

        logger.info(
            f"Phase 1 complete: {len(cpp.chunk_graph.chunks)} chunks generated"
        )

        # Validate chunk count
        actual_count = len(cpp.chunk_graph.chunks)
        if actual_count != EXPECTED_CHUNK_COUNT:
            logger.warning(
                f"Expected {EXPECTED_CHUNK_COUNT} chunks, got {actual_count}. "
                f"PA×DIM structure may be incomplete."
            )

        return cpp


__all__ = [
    "Phase1SPCIngestionContract",
    "PHASE1_VERSION",
    "EXPECTED_CHUNK_COUNT",
    "POLICY_AREAS",
    "DIMENSIONS",
    "SubfaseMetadata",
]

===== FILE: src/saaaaaa/core/phases/phase1_to_phase2_adapter/__init__.py =====
"""
Phase 1 → Phase 2 Adapter Contract
===================================

This module implements the adapter contract that transforms CanonPolicyPackage
(Phase 1 output) into PreprocessedDocument (Phase 2 input).

Responsibilities:
-----------------
1. Convert 60 PA×DIM chunks to sentences
2. Preserve chunk_id, policy_area_id, dimension_id in sentence_metadata.extra
3. Maintain chunk graph edges
4. Preserve all facets (policy, time, geo, entity, budget, KPI)
5. Validate preservation of critical metadata

Input Contract:
---------------
CanonPolicyPackage (from Phase 1):
    - chunk_graph with 60 chunks
    - Each chunk has policy_area_id (PA01-PA10)
    - Each chunk has dimension_id (DIM01-DIM06)
    - policy_manifest, quality_metrics, integrity_index

Output Contract:
----------------
PreprocessedDocument (for Phase 2):
    - sentences: tuple[str] (one per chunk)
    - sentence_metadata: tuple[SentenceMetadata]
    - sentence_metadata[i].extra MUST contain:
        - chunk_id: str
        - policy_area_id: str (PA01-PA10)
        - dimension_id: str (DIM01-DIM06)
        - resolution: str
        - policy_facets: dict
```

```
        - time_facets: dict
        - geo_facets: dict
    - metadata: dict with quality_metrics, policy_manifest

Invariants:
-----------
1. len(sentences) == 60 (one per chunk)
2. All sentence_metadata have chunk_id in extra
3. All sentence_metadata have policy_area_id in extra
4. All sentence_metadata have dimension_id in extra
5. processing_mode == "chunked"

Author: F.A.R.F.A.N Architecture Team
Date: 2025-01-19
"""

from __future__ import annotations

import logging
from typing import Any

from saaaaaa.core.orchestrator.core import PreprocessedDocument
from saaaaaa.core.phases.phase_protocol import (
    ContractValidationResult,
    PhaseContract,
)
from saaaaaa.processing.cpp_ingestion.models import CanonPolicyPackage

logger = logging.getLogger(__name__)


def _meta_extra(meta: Any) -> dict[str, Any]:
    """Extract extra metadata from sentence metadata entries."""
    if isinstance(meta, dict):
        return meta.get("extra", {}) or {}
    if hasattr(meta, "extra"):
        return getattr(meta, "extra") or {}
    return {}


class AdapterContract(PhaseContract[CanonPolicyPackage, PreprocessedDocument]):
    """
    Adapter contract enforcing PA×DIM metadata preservation.

    This contract ensures that the transformation from CanonPolicyPackage
    to PreprocessedDocument preserves all critical chunk metadata needed
    for Phase 2 question routing.
    """

    def __init__(self):
        """Initialize adapter contract with invariants."""
        super().__init__(phase_name="phase1_to_phase2_adapter")

        # Invariant: All chunks preserved as sentences
        self.add_invariant(
            name="chunk_count_preserved",
            description="All chunks must be preserved as sentences",
            check=lambda data: len(data.sentences) > 0,
            error_message="No sentences in PreprocessedDocument",
        )

        # Invariant: Processing mode is chunked
        self.add_invariant(
            name="processing_mode_chunked",
            description="processing_mode must be 'chunked'",
            check=lambda data: data.processing_mode == "chunked",
            error_message="processing_mode must be 'chunked' for SPC adapter",
        )
```

```
        # Invariant: All sentence_metadata have chunk_id
        self.add_invariant(
            name="chunk_id_preserved",
            description="All sentence_metadata must have chunk_id in extra",
            check=lambda data: all("chunk_id" in _meta_extra(meta) for meta in
data.sentence_metadata),
            error_message="Missing chunk_id in sentence_metadata.extra",
        )

        # Invariant: All sentence_metadata have policy_area_id
        self.add_invariant(
            name="policy_area_id_preserved",
            description="All sentence_metadata must have policy_area_id in extra",
            check=lambda data: all("policy_area_id" in _meta_extra(meta) for meta in
data.sentence_metadata),
            error_message="Missing policy_area_id in sentence_metadata.extra - CRITICAL
for Phase 2",
        )

        # Invariant: All sentence_metadata have dimension_id
        self.add_invariant(
            name="dimension_id_preserved",
            description="All sentence_metadata must have dimension_id in extra",
            check=lambda data: all("dimension_id" in _meta_extra(meta) for meta in
data.sentence_metadata),
            error_message="Missing dimension_id in sentence_metadata.extra - CRITICAL for
Phase 2",
        )

    def validate_input(self, input_data: Any) -> ContractValidationResult:
        """
        Validate CanonPolicyPackage input.

        Args:
            input_data: Input to validate

        Returns:
            ContractValidationResult
        """
        errors = []
        warnings = []

        # Type check
        if not isinstance(input_data, CanonPolicyPackage):
            errors.append(
                f"Expected CanonPolicyPackage, got {type(input_data).__name__}"
            )
            return ContractValidationResult(
                passed=False,
                contract_type="input",
                phase_name=self.phase_name,
                errors=errors,
            )

        # Validate chunk_graph exists
        if not hasattr(input_data, 'chunk_graph') or not input_data.chunk_graph:
            errors.append("CanonPolicyPackage missing chunk_graph")

        # Validate chunks exist
        if hasattr(input_data, 'chunk_graph') and input_data.chunk_graph:
            chunk_count = len(input_data.chunk_graph.chunks)
            if chunk_count == 0:
                errors.append("chunk_graph.chunks is empty")
            elif chunk_count != 60:
                warnings.append(
                    f"Expected 60 chunks (10 PA × 6 DIM), got {chunk_count}"
                )
```

```
        # Validate PA×DIM tags present
        missing_pa_dim = []
        for chunk_id, chunk in input_data.chunk_graph.chunks.items():
            if not hasattr(chunk, 'policy_area_id') or not chunk.policy_area_id:
                missing_pa_dim.append(f"{chunk_id}: missing policy_area_id")
            if not hasattr(chunk, 'dimension_id') or not chunk.dimension_id:
                missing_pa_dim.append(f"{chunk_id}: missing dimension_id")

        if missing_pa_dim:
            errors.append(
                f"Chunks missing PA×DIM tags: {missing_pa_dim[:5]}"
            )

    return ContractValidationResult(
        passed=len(errors) == 0,
        contract_type="input",
        phase_name=self.phase_name,
        errors=errors,
        warnings=warnings,
    )

def validate_output(self, output_data: Any) -> ContractValidationResult:
    """
    Validate PreprocessedDocument output.

    Args:
        output_data: Output to validate

    Returns:
        ContractValidationResult
    """
    errors = []
    warnings = []

    # Type check
    if not isinstance(output_data, PreprocessedDocument):
        errors.append(
            f"Expected PreprocessedDocument, got {type(output_data).__name__}"
        )
        return ContractValidationResult(
            passed=False,
            contract_type="output",
            phase_name=self.phase_name,
            errors=errors,
        )

    # Validate sentences exist
    if not hasattr(output_data, 'sentences') or not output_data.sentences:
        errors.append("PreprocessedDocument.sentences is empty")

    # Validate processing_mode
    if not hasattr(output_data, 'processing_mode') or output_data.processing_mode !=
"chunked":
        errors.append(
            f"processing_mode must be 'chunked', got '{getattr(output_data,
'processing_mode', None)}'"
        )

    # Validate sentence_metadata exists and matches sentences
    if hasattr(output_data, 'sentences') and hasattr(output_data,
'sentence_metadata'):
        if len(output_data.sentence_metadata) != len(output_data.sentences):
            errors.append(
                f"sentence_metadata count ({len(output_data.sentence_metadata)}) != "
                f"sentences count ({len(output_data.sentences)})"
            )
```

```
        # Validate PA×DIM preservation in sentence_metadata
        missing_metadata = []
        for idx, meta in enumerate(output_data.sentence_metadata):
            extra = _meta_extra(meta)
            if not extra:
                missing_metadata.append(f"sentence[{idx}]: no extra field")
                continue

            if 'chunk_id' not in extra:
                missing_metadata.append(f"sentence[{idx}]: missing chunk_id")
            if 'policy_area_id' not in extra:
                missing_metadata.append(f"sentence[{idx}]: missing policy_area_id")
            if 'dimension_id' not in extra:
                missing_metadata.append(f"sentence[{idx}]: missing dimension_id")

        if missing_metadata:
            errors.append(
                f"Metadata preservation failed: {missing_metadata[:5]}"
            )

    return ContractValidationResult(
        passed=len(errors) == 0,
        contract_type="output",
        phase_name=self.phase_name,
        errors=errors,
        warnings=warnings,
    )

async def execute(self, input_data: CanonPolicyPackage) -> PreprocessedDocument:
    """
    Execute adapter transformation.

    Args:
        input_data: CanonPolicyPackage from Phase 1

    Returns:
        PreprocessedDocument for Phase 2

    Raises:
        ImportError: If SPCAdapter not available
        ValueError: If transformation fails
    """
    logger.info(f"Starting adapter: CanonPolicyPackage → PreprocessedDocument")

    # Use existing SPCAdapter implementation
    from saaaaaa.utils.spc_adapter import SPCAdapter

    adapter = SPCAdapter(enable_runtime_validation=False)  # We validate here

    # Get document_id from metadata
    document_id = input_data.metadata.get('document_id', 'unknown')

    # Transform
    preprocessed = adapter.to_preprocessed_document(
        input_data,
        document_id=document_id
    )

    logger.info(
        f"Adapter complete: {len(preprocessed.sentences)} sentences, "
        f"mode={preprocessed.processing_mode}"
    )

    return preprocessed


__all__ = [
    "AdapterContract",
```

]

===== FILE: src/saaaaaa/core/phases/phase2_types.py =====

```python
"""
Types for Phase 2 (Microquestions)
==================================

This module defines the canonical data structures for the output of
Phase 2, which involves processing the PreprocessedDocument to generate
a series of microquestions and their answers.

These types are used by the PhaseOrchestrator to validate and record
the results of the core orchestrator's execution.

Author: F.A.R.F.A.N Architecture Team
Date: 2025-11-21
"""

from __future__ import annotations

from dataclasses import dataclass
from typing import Any, List, Tuple


@dataclass
class Phase2QuestionResult:
    """
    Represents the result for a single microquestion generated
    and answered during Phase 2.
    """

    base_slot: str
    question_id: str
    question_global: int | None
    policy_area_id: str | None = None
    dimension_id: str | None = None
    cluster_id: str | None = None
    evidence: dict[str, Any] | None = None
    validation: dict[str, Any] | None = None
    trace: dict[str, Any] | None = None
    metadata: dict[str, Any] | None = None
    human_readable_output: str | None = None  # Added for v3 contracts


@dataclass
class Phase2Result:
    """
    Represents the complete output of Phase 2, containing all
    the generated microquestions.
    """

    questions: List[Phase2QuestionResult]


def _extract_questions(result: Any) -> tuple[List[Any] | None, list[str]]:
    """Normalize Phase 2 result into a list of question dicts if possible."""
    errors: list[str] = []
    if result is None:
        errors.append("Phase 2 returned no data")
        return None, errors

    if isinstance(result, dict) and "questions" in result:
        questions = result.get("questions")
    elif hasattr(result, "questions"):
        questions = getattr(result, "questions")
    else:
        errors.append("Phase 2 result missing 'questions'")
        return None, errors
```

```python
    if not isinstance(questions, list):
        errors.append("Phase 2 questions must be a list")
        return None, errors

    return questions, errors


def validate_phase2_result(result: Any) -> Tuple[bool, list[str], List[dict[str, Any]] |
None]:
    """
    Validate the structure of a Phase 2 result.

    Returns:
        Tuple of (is_valid, errors, normalized_questions)
    """
    questions, errors = _extract_questions(result)
    if questions is None:
        return False, errors, None

    if len(questions) == 0:
        errors.append("Phase 2 questions list is empty or missing")
        return False, errors, questions

    normalized: list[dict[str, Any]] = []
    for idx, q in enumerate(questions):
        if not isinstance(q, dict):
            errors.append(f"Question {idx} must be a dict")
            continue

        required_keys = ["base_slot", "question_id", "question_global", "evidence",
"validation"]
        missing = [key for key in required_keys if q.get(key) is None]
        if missing:
            errors.append(f"Question {idx} missing keys: {', '.join(missing)}")

        normalized.append(q)

    return len(errors) == 0, errors, normalized


__all__ = [
    "Phase2QuestionResult",
    "Phase2Result",
    "validate_phase2_result",
]

===== FILE: src/saaaaaa/core/phases/phase_orchestrator.py =====
"""
Phase Orchestrator - Constitutional Sequence Enforcement
============================================================

This module implements the PhaseOrchestrator which GUARANTEES that:

1. Phases execute in STRICT sequence (0 → 1 → Adapter → 2)
2. Each phase's output becomes the NEXT phase's input
3. NO phase can be bypassed
4. ALL contracts are validated at boundaries
5. ALL invariants are checked
6. FULL traceability in manifest

The orchestrator is the SINGLE point of entry for pipeline execution.
It is IMPOSSIBLE to run phases out of order or skip validation.

Design Principles:
------------------
- **Single Entry Point**: Only `run_pipeline()` executes the full sequence
- **No Bypass**: Phases cannot be called directly from outside
```

```python
- **Contract Enforcement**: All inputs/outputs validated
- **Deterministic**: Same Phase0Input → same outputs
- **Auditable**: Full manifest with all phase boundaries

Phase Sequence (IMMUTABLE):
---------------------------
Phase 0: input_validation
    Input: Phase0Input (pdf_path, run_id, questionnaire_path)
    Output: CanonicalInput
    ↓
Phase 1: spc_ingestion
    Input: CanonicalInput
    Output: CanonPolicyPackage
    ↓
Adapter: phase1_to_phase2
    Input: CanonPolicyPackage
    Output: PreprocessedDocument
    ↓
Phase 2: microquestions
    Input: PreprocessedDocument
    Output: Phase2Result

Author: F.A.R.F.A.N Architecture Team
Date: 2025-01-19
"""

from __future__ import annotations

import logging
from dataclasses import dataclass, field
from pathlib import Path
from typing import Any

from saaaaaa.core.orchestrator.factory import build_processor
from saaaaaa.core.phases.phase_protocol import (
    ContractValidationResult,
    PhaseManifestBuilder,
    PhaseMetadata,
)
from saaaaaa.core.phases.phase0_input_validation import (
    CanonicalInput,
    Phase0Input,
    Phase0ValidationContract,
)
from saaaaaa.core.phases.phase1_spc_ingestion import (
    Phase1SPCIngestionContract,
)
from saaaaaa.core.phases.phase2_types import validate_phase2_result

logger = logging.getLogger(__name__)


@dataclass
class PipelineResult:
    """
    Complete result of pipeline execution.

    This is the ONLY output of PhaseOrchestrator.run_pipeline().
    """

    success: bool
    run_id: str

    # Phase outputs (populated if phase succeeded)
    canonical_input: CanonicalInput | None = None
    canon_policy_package: Any | None = None  # CanonPolicyPackage
    preprocessed_document: Any | None = None  # PreprocessedDocument
    phase2_result: Any | None = None  # Phase2Result
```

````python
    # Execution metadata
    phases_completed: int = 0
    phases_failed: int = 0
    total_duration_ms: float = 0.0

    # Error tracking
    errors: list[str] = field(default_factory=list)

    # Manifest
    manifest: dict[str, Any] = field(default_factory=dict)


class PhaseOrchestrator:
    """
    Orchestrator that enforces the canonical phase sequence.

    This class is the CONSTITUTIONAL GUARANTEE that phases execute
    in order with full contract validation.

    Usage:
    ------
    ```python
    orchestrator = PhaseOrchestrator()
    result = await orchestrator.run_pipeline(
        pdf_path=Path("plan.pdf"),
        run_id="plan1",
        questionnaire_path=Path("questionnaire.json"),
        artifacts_dir=Path("artifacts/plan1"),
    )

    if result.success:
        print(f"Pipeline succeeded: {result.phases_completed} phases")
    else:
        print(f"Pipeline failed: {result.errors}")
    ```
    """

    def __init__(self):
        """Initialize orchestrator with phase contracts."""
        logger.info("Initializing PhaseOrchestrator with constitutional constraints")

        # Initialize phase contracts
        self.phase0 = Phase0ValidationContract()
        self.phase1 = Phase1SPCIngestionContract()

        # Import and initialize adapter contract
        from saaaaaa.core.phases.phase1_to_phase2_adapter import AdapterContract
        self.adapter = AdapterContract()

        # self.phase2 = Phase2Contract()    # To be implemented

        # Initialize manifest builder
        self.manifest_builder = PhaseManifestBuilder()

        logger.info("PhaseOrchestrator initialized successfully")

    async def run_pipeline(
        self,
        pdf_path: Path,
        run_id: str,
        questionnaire_path: Path | None = None,
        artifacts_dir: Path | None = None,
    ) -> PipelineResult:
        """
        Execute the COMPLETE canonical pipeline in STRICT sequence.

        This is the ONLY way to run the pipeline. It enforces:
````

```
1. Phase 0 → Phase 1 → Adapter → Phase 2
2. Contract validation at ALL boundaries
3. Invariant checking for ALL phases
4. Full manifest generation

Args:
    pdf_path: Path to input PDF
    run_id: Unique run identifier
    questionnaire_path: Optional questionnaire path
    artifacts_dir: Optional directory for artifacts

Returns:
    PipelineResult with success status and all phase outputs

Raises:
    This method does NOT raise exceptions. All errors are captured
    in PipelineResult.errors and PipelineResult.success = False.
"""
logger.info(f"Starting pipeline execution: run_id={run_id}")

result = PipelineResult(
    success=False,  # Will be set to True only if ALL phases succeed
    run_id=run_id,
)

# Create artifacts directory if provided
if artifacts_dir:
    artifacts_dir.mkdir(parents=True, exist_ok=True)

try:
    # ======================================================================
    # PHASE 0: Input Validation
    # ======================================================================
    logger.info("=" * 70)
    logger.info("PHASE 0: Input Validation")
    logger.info("=" * 70)

    phase0_input = Phase0Input(
        pdf_path=pdf_path,
        run_id=run_id,
        questionnaire_path=questionnaire_path,
    )

    canonical_input, phase0_metadata = await self.phase0.run(phase0_input)

    # Record Phase 0 in manifest
    self.manifest_builder.record_phase(
        phase_name="phase0_input_validation",
        metadata=phase0_metadata,
        input_validation=self.phase0.validate_input(phase0_input),
        output_validation=self.phase0.validate_output(canonical_input),
        invariants_checked=[inv.name for inv in self.phase0.invariants],
        artifacts=[],  # No artifacts for Phase 0
    )

    result.canonical_input = canonical_input
    result.phases_completed += 1
    result.total_duration_ms += phase0_metadata.duration_ms or 0.0

    logger.info(
        f"Phase 0 completed successfully in {phase0_metadata.duration_ms:.0f}ms"
    )

    # ======================================================================
    # PHASE 1: SPC Ingestion
    # ======================================================================
    logger.info("=" * 70)
    logger.info("PHASE 1: SPC Ingestion (15 subfases)")
```

```python
logger.info("=" * 70)

# Phase 1 input is Phase 0 output (guaranteed by type system)
cpp, phase1_metadata = await self.phase1.run(canonical_input)

# Record Phase 1 in manifest
self.manifest_builder.record_phase(
    phase_name="phase1_spc_ingestion",
    metadata=phase1_metadata,
    input_validation=self.phase1.validate_input(canonical_input),
    output_validation=self.phase1.validate_output(cpp),
    invariants_checked=[inv.name for inv in self.phase1.invariants],
    artifacts=[],  # Artifacts tracked separately
)

result.canon_policy_package = cpp
result.phases_completed += 1
result.total_duration_ms += phase1_metadata.duration_ms or 0.0

logger.info(
    f"Phase 1 completed successfully in {phase1_metadata.duration_ms:.0f}ms"
)
logger.info(f"Generated {len(cpp.chunk_graph.chunks)} chunks")

# ====================================================================
# ADAPTER: Phase 1 → Phase 2
# ====================================================================
logger.info("=" * 70)
logger.info("ADAPTER: CanonPolicyPackage → PreprocessedDocument")
logger.info("=" * 70)

# Run adapter with contract enforcement
preprocessed, adapter_metadata = await self.adapter.run(cpp)

# Record Adapter in manifest
self.manifest_builder.record_phase(
    phase_name="phase1_to_phase2_adapter",
    metadata=adapter_metadata,
    input_validation=self.adapter.validate_input(cpp),
    output_validation=self.adapter.validate_output(preprocessed),
    invariants_checked=[inv.name for inv in self.adapter.invariants],
    artifacts=[],
)

result.preprocessed_document = preprocessed
result.phases_completed += 1
result.total_duration_ms += adapter_metadata.duration_ms or 0.0

logger.info(
    f"Adapter completed successfully in {adapter_metadata.duration_ms:.0f}ms"
)
logger.info(
    f"PreprocessedDocument: {len(preprocessed.sentences)} sentences, "
    f"mode={preprocessed.processing_mode}"
)

# ====================================================================
# CORE ORCHESTRATOR: Phases 0-10 (Includes Micro-Questions)
# ====================================================================
logger.info("=" * 70)
logger.info("CORE ORCHESTRATOR: Executing Phases 0-10")
logger.info("=" * 70)

# --- Imports for Phase 2 Integration ---
from datetime import datetime, timedelta, timezone

# --- Execute Core Orchestrator ---
processor = build_processor()
```

```
        p2_block_started_at = datetime.now(timezone.utc)
        core_results = await processor.orchestrator.process_development_plan_async(
            pdf_path=str(pdf_path),
            preprocessed_document=preprocessed,
        )
        p2_block_finished_at = datetime.now(timezone.utc)

        # --- Process and Record Phase 2 ---
        phase2_success = False
        phase2_errors: list[str] = []
        phase2_questions: list[dict[str, Any]] | None = None

        if len(core_results) >= 3:
            phase2_core = core_results[2]  # FASE 2 - Micro Preguntas
            result.phase2_result = phase2_core.data if phase2_core.success else None

            if phase2_core.success:
                is_valid, validation_errors, normalized_questions =
validate_phase2_result(
                    phase2_core.data
                )
                phase2_questions = normalized_questions
                if not is_valid:
                    phase2_errors.extend(validation_errors)
                    phase2_errors.append(
                        "Phase 2 failed structural invariant: questions list is empty
or missing."
                    )
                phase2_success = phase2_core.success and is_valid
            else:
                phase2_errors.append(
                    f"Core phase 2 returned error: {phase2_core.error}"
                )

            # --- Create Manifest Entry for Phase 2 ---
            p2_error_msg = "; ".join(phase2_errors) if phase2_errors else None

            # Approximate start/end times for the manifest metadata
            p2_duration = timedelta(milliseconds=phase2_core.duration_ms)
            p2_started_at_approx = p2_block_finished_at - p2_duration

            p2_metadata = PhaseMetadata(
                phase_name="phase2_microquestions",
                success=phase2_success,
                error=p2_error_msg,
                duration_ms=phase2_core.duration_ms,
                started_at=p2_started_at_approx.isoformat(),
                finished_at=p2_block_finished_at.isoformat(),
            )

            # Create validation results to satisfy the manifest builder
            dummy_input_validation = ContractValidationResult(
                passed=True,
                contract_type="input",
                phase_name="phase2_microquestions",
            )
            dummy_output_validation = ContractValidationResult(
                passed=phase2_success,
                contract_type="output",
                phase_name="phase2_microquestions",
                errors=phase2_errors,
            )

            self.manifest_builder.record_phase(
                phase_name="phase2_microquestions",
                metadata=p2_metadata,
                input_validation=dummy_input_validation,
                output_validation=dummy_output_validation,
```

```
                invariants_checked=["questions_are_present_and_non_empty"],
                artifacts=[],
            )
            self.manifest_builder.phases["phase2_microquestions"]["question_count"] =
len(phase2_questions or [])
            if phase2_errors:
                self.manifest_builder.phases["phase2_microquestions"]["errors"] =
list(phase2_errors)

            if not phase2_success:
                error_msg = f"Core Orchestrator Phase 2 failed: {p2_error_msg}"
                logger.error(error_msg)
                result.errors.append(error_msg)
                result.phases_failed += 1
            else:
                # Only add core result count if Phase 2 was successful
                result.phase2_result = {"questions": phase2_questions or []}
                result.phases_completed += len(core_results)
                logger.info(
                    f"Core Orchestrator completed {len(core_results)} phases
successfully"
                )

        else:
            # Phase 2 was not even present in the results
            missing_p2_error = "Core Orchestrator did not produce a result for Phase
2."
            logger.error(missing_p2_error)
            result.errors.append(missing_p2_error)
            result.phases_failed += 1
            # Create a failure record in the manifest
            p2_metadata = PhaseMetadata(
                phase_name="phase2_microquestions",
                success=False,
                error=missing_p2_error,
                started_at=p2_block_started_at.isoformat(),
                finished_at=p2_block_finished_at.isoformat(),
                duration_ms=(p2_block_finished_at-p2_block_started_at).total_seconds()
 * 1000,
            )
            self.manifest_builder.record_phase(
                phase_name="phase2_microquestions",
                metadata=p2_metadata,
                input_validation=ContractValidationResult(passed=False,
contract_type="input", phase_name="phase2_microquestions", errors=[missing_p2_error]),
                output_validation=ContractValidationResult(passed=False,
contract_type="output", phase_name="phase2_microquestions", errors=[missing_p2_error]),
                invariants_checked=[],
                artifacts=[],
            )
            self.manifest_builder.phases["phase2_microquestions"]["question_count"] =
0
            self.manifest_builder.phases["phase2_microquestions"]["errors"] =
[missing_p2_error]


        # ======================================================================
        # PIPELINE SUCCESS
        # ======================================================================
        # Success is now conditional on all canonical phases, including Phase 2
        all_phases_ok = all(
            p.get("status") == "success"
            for p in self.manifest_builder.phases.values()
        )

        if all_phases_ok:
            result.success = True
            logger.info("=" * 70)
```

```
            logger.info(f"PIPELINE COMPLETED SUCCESSFULLY")
            logger.info(f"Phases completed: {result.phases_completed}")
            logger.info(f"Total duration: {result.total_duration_ms:.0f}ms")
            logger.info("=" * 70)
        else:
            # Ensure result.success is False if we got here with a failure
            result.success = False
            final_error = f"Pipeline failed. Check manifest for details. Completed:
{result.phases_completed}, Failed: {result.phases_failed}"
            if not result.errors:
                result.errors.append(final_error)
            logger.error(final_error)

    except Exception as e:
        # Capture error
        error_msg = f"Pipeline failed: {e}"
        logger.error(error_msg, exc_info=True)
        result.errors.append(error_msg)
        result.success = False
        result.phases_failed += 1

    finally:
        # Always generate manifest
        result.manifest = self.manifest_builder.to_dict()
        phase2_entry = result.manifest.get("phases", {}).get("phase2_microquestions")
        if phase2_entry is not None:
            result.manifest["phases"]["phase2"] = phase2_entry

        # Save manifest if artifacts_dir provided
        if artifacts_dir:
            manifest_path = artifacts_dir / "phase_manifest.json"
            self.manifest_builder.save(manifest_path)
            logger.info(f"Phase manifest saved to {manifest_path}")

    return result


__all__ = [
    "PhaseOrchestrator",
    "PipelineResult",
]

===== FILE: src/saaaaaa/core/phases/phase_protocol.py =====
"""
Phase Contract Protocol - Constitutional Constraint System
==============================================================

This module implements the constitutional constraint framework where each phase:

1. Has an EXPLICIT input contract (typed, validated)
2. Has an EXPLICIT output contract (typed, validated)
3. Communicates ONLY through these contracts (no side channels)
4. Is enforced by validators (runtime contract checking)
5. Is tracked in the verification manifest (full traceability)

Design Principles:
------------------
- **Single Entry Point**: Each phase accepts exactly ONE input type
- **Single Exit Point**: Each phase produces exactly ONE output type
- **No Bypass**: The orchestrator enforces sequential execution
- **Verifiable**: All contracts are validated and logged
- **Deterministic**: Same input → same output (modulo controlled randomness)

Phase Structure:
----------------
phase0_input_validation:
    Input: Phase0Input (raw PDF path + run_id)
    Output: CanonicalInput (validated, hashed, ready)
```

```
phase1_spc_ingestion:
    Input: CanonicalInput
    Output: CanonPolicyPackage (60 chunks, PA×DIM structured)

phase1_to_phase2_adapter:
    Input: CanonPolicyPackage
    Output: PreprocessedDocument (chunked mode)

phase2_microquestions:
    Input: PreprocessedDocument
    Output: Phase2Result (305 questions answered)

Author: F.A.R.F.A.N Architecture Team
Date: 2025-01-19
"""

from __future__ import annotations

import hashlib
import json
from abc import ABC, abstractmethod
from dataclasses import asdict, dataclass, field
from datetime import datetime, timezone
from pathlib import Path
from typing import Any, Generic, TypeVar

from pydantic import BaseModel, Field, ValidationError

# Type variables for generic phase contracts
TInput = TypeVar("TInput")
TOutput = TypeVar("TOutput")


@dataclass
class PhaseInvariant:
    """An invariant that must hold for a phase."""

    name: str
    description: str
    check: callable  # Function that returns bool
    error_message: str


@dataclass
class PhaseMetadata:
    """Metadata for a phase execution."""

    phase_name: str
    started_at: str
    finished_at: str | None = None
    duration_ms: float | None = None
    success: bool = False
    error: str | None = None


@dataclass
class ContractValidationResult:
    """Result of validating a contract."""

    passed: bool
    contract_type: str  # "input" or "output"
    phase_name: str
    errors: list[str] = field(default_factory=list)
    warnings: list[str] = field(default_factory=list)
    validation_timestamp: str = field(
        default_factory=lambda: datetime.now(timezone.utc).isoformat()
    )
```

```python
class PhaseContract(ABC, Generic[TInput, TOutput]):
    """
    Abstract base class for phase contracts.

    Each phase must implement:
    1. Input contract validation
    2. Output contract validation
    3. Invariant checking
    4. Phase execution logic

    This enforces the constitutional constraint that phases communicate
    ONLY through validated contracts.
    """

    def __init__(self, phase_name: str):
        """
        Initialize phase contract.

        Args:
            phase_name: Canonical name of the phase (e.g., "phase0_input_validation")
        """
        self.phase_name = phase_name
        self.invariants: list[PhaseInvariant] = []
        self.metadata: PhaseMetadata | None = None

    @abstractmethod
    def validate_input(self, input_data: Any) -> ContractValidationResult:
        """
        Validate input contract.

        Args:
            input_data: Input to validate

        Returns:
            ContractValidationResult with validation status
        """
        pass

    @abstractmethod
    def validate_output(self, output_data: Any) -> ContractValidationResult:
        """
        Validate output contract.

        Args:
            output_data: Output to validate

        Returns:
            ContractValidationResult with validation status
        """
        pass

    @abstractmethod
    async def execute(self, input_data: TInput) -> TOutput:
        """
        Execute the phase logic.

        Args:
            input_data: Validated input conforming to input contract

        Returns:
            Output conforming to output contract

        Raises:
            ValueError: If input contract validation fails
            RuntimeError: If phase execution fails
        """
```

```
    pass

def add_invariant(
    self,
    name: str,
    description: str,
    check: callable,
    error_message: str,
) -> None:
    """
    Add an invariant to this phase.

    Args:
        name: Invariant name
        description: Human-readable description
        check: Function that returns bool (True = invariant holds)
        error_message: Error message if invariant fails
    """
    self.invariants.append(
        PhaseInvariant(
            name=name,
            description=description,
            check=check,
            error_message=error_message,
        )
    )

def check_invariants(self, data: Any) -> tuple[bool, list[str]]:
    """
    Check all invariants for this phase.

    Args:
        data: Data to check invariants against

    Returns:
        Tuple of (all_passed, failed_invariant_messages)
    """
    failed_messages = []
    for inv in self.invariants:
        try:
            if not inv.check(data):
                failed_messages.append(f"{inv.name}: {inv.error_message}")
        except Exception as e:
            failed_messages.append(f"{inv.name}: Exception during check: {e}")

    return len(failed_messages) == 0, failed_messages

async def run(self, input_data: TInput) -> tuple[TOutput, PhaseMetadata]:
    """
    Run the complete phase with validation and invariant checking.

    This is the ONLY way to execute a phase - it enforces:
    1. Input validation
    2. Invariant checking (pre-execution if applicable)
    3. Phase execution
    4. Output validation
    5. Invariant checking (post-execution)
    6. Metadata recording

    Args:
        input_data: Input to the phase

    Returns:
        Tuple of (output_data, phase_metadata)

    Raises:
        ValueError: If contract validation fails
        RuntimeError: If invariants fail or execution fails
```

```
        """
        started_at = datetime.now(timezone.utc)
        metadata = PhaseMetadata(
            phase_name=self.phase_name,
            started_at=started_at.isoformat(),
        )

        try:
            # 1. Validate input contract
            input_validation = self.validate_input(input_data)
            if not input_validation.passed:
                error_msg = f"Input contract validation failed: {input_validation.errors}"
                metadata.error = error_msg
                metadata.success = False
                raise ValueError(error_msg)

            # 2. Execute phase
            output_data = await self.execute(input_data)

            # 3. Validate output contract
            output_validation = self.validate_output(output_data)
            if not output_validation.passed:
                error_msg = f"Output contract validation failed:
{output_validation.errors}"
                metadata.error = error_msg
                metadata.success = False
                raise ValueError(error_msg)

            # 4. Check invariants
            invariants_passed, failed_invariants = self.check_invariants(output_data)
            if not invariants_passed:
                error_msg = f"Phase invariants failed: {failed_invariants}"
                metadata.error = error_msg
                metadata.success = False
                raise RuntimeError(error_msg)

            # Success
            metadata.success = True
            return output_data, metadata

        except Exception as e:
            metadata.error = str(e)
            metadata.success = False
            raise

        finally:
            finished_at = datetime.now(timezone.utc)
            metadata.finished_at = finished_at.isoformat()
            metadata.duration_ms = (
                finished_at - started_at
            ).total_seconds() * 1000
            self.metadata = metadata


@dataclass
class PhaseArtifact:
    """An artifact produced by a phase."""

    artifact_name: str
    artifact_path: Path
    sha256: str
    size_bytes: int
    created_at: str


class PhaseManifestBuilder:
    """
    Builds the phase-explicit section of the verification manifest.
```

```
Each phase execution is recorded with:
- Input/output contract hashes
- Invariants checked
- Artifacts produced
- Timing information
"""

def __init__(self):
    """Initialize manifest builder."""
    self.phases: dict[str, dict[str, Any]] = {}

def record_phase(
    self,
    phase_name: str,
    metadata: PhaseMetadata,
    input_validation: ContractValidationResult,
    output_validation: ContractValidationResult,
    invariants_checked: list[str],
    artifacts: list[PhaseArtifact],
) -> None:
    """
    Record a phase execution in the manifest.

    Args:
        phase_name: Name of the phase
        metadata: Phase execution metadata
        input_validation: Input contract validation result
        output_validation: Output contract validation result
        invariants_checked: List of invariant names that were checked
        artifacts: List of artifacts produced by this phase
    """
    self.phases[phase_name] = {
        "status": "success" if metadata.success else "failed",
        "started_at": metadata.started_at,
        "finished_at": metadata.finished_at,
        "duration_ms": metadata.duration_ms,
        "input_contract": {
            "validation_passed": input_validation.passed,
            "errors": input_validation.errors,
            "warnings": input_validation.warnings,
        },
        "output_contract": {
            "validation_passed": output_validation.passed,
            "errors": output_validation.errors,
            "warnings": output_validation.warnings,
        },
        "invariants_checked": invariants_checked,
        "invariants_satisfied": metadata.success,
        "artifacts": [
            {
                "name": a.artifact_name,
                "path": str(a.artifact_path),
                "sha256": a.sha256,
                "size_bytes": a.size_bytes,
            }
            for a in artifacts
        ],
        "error": metadata.error,
    }

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

    Returns:
        Dictionary representation of the phase manifest
    """
```

```python
        return {
            "phases": self.phases,
            "total_phases": len(self.phases),
            "successful_phases": sum(
                1 for p in self.phases.values() if p["status"] == "success"
            ),
            "failed_phases": sum(
                1 for p in self.phases.values() if p["status"] == "failed"
            ),
        }

    def save(self, output_path: Path) -> None:
        """
        Save manifest to JSON file.

        Args:
            output_path: Path to save manifest
        """
        with open(output_path, "w") as f:
            json.dump(self.to_dict(), f, indent=2)


def compute_contract_hash(contract_data: Any) -> str:
    """
    Compute SHA256 hash of a contract's data.

    Args:
        contract_data: Contract data (dict, dataclass, or Pydantic model)

    Returns:
        Hex-encoded SHA256 hash
    """
    # Convert to dict if needed
    if hasattr(contract_data, "dict"):
        # Pydantic model
        data_dict = contract_data.dict()
    elif hasattr(contract_data, "__dataclass_fields__"):
        # Dataclass
        data_dict = asdict(contract_data)
    elif isinstance(contract_data, dict):
        data_dict = contract_data
    else:
        raise TypeError(f"Cannot hash contract data of type {type(contract_data)}")

    # Serialize to JSON with sorted keys for determinism
    json_str = json.dumps(data_dict, sort_keys=True, separators=(",", ":"))
    return hashlib.sha256(json_str.encode("utf-8")).hexdigest()


__all__ = [
    "PhaseContract",
    "PhaseInvariant",
    "PhaseMetadata",
    "ContractValidationResult",
    "PhaseArtifact",
    "PhaseManifestBuilder",
    "compute_contract_hash",
]
```

===== FILE: src/saaaaaa/core/ports.py =====

```python
"""
Port interfaces for dependency injection.

Ports define abstract interfaces for external interactions (I/O, time, environment).
These are implemented by adapters in the infrastructure layer.

This follows the Ports and Adapters (Hexagonal) architecture pattern:
- Ports are in the core layer (no dependencies)
```

```
- Adapters are in the infrastructure layer (can import anything)
- Core modules depend on ports (abstractions), not adapters (implementations)

Version: 1.0.0
"""

from datetime import datetime
from typing import Any, Protocol


class FilePort(Protocol):
    """Port for file system operations.

    Implementations provide access to file reading and writing.
    Core modules receive a FilePort instance via dependency injection.
    """

    def read_text(self, path: str, encoding: str = "utf-8") -> str:
        """Read text from a file.

        Args:
            path: File path to read
            encoding: Text encoding (default: utf-8)

        Returns:
            File contents as string

        Raises:
            FileNotFoundError: If file does not exist
            PermissionError: If file cannot be read
        """
        ...

    def write_text(self, path: str, content: str, encoding: str = "utf-8") -> None:
        """Write text to a file.

        Args:
            path: File path to write
            content: Text content to write
            encoding: Text encoding (default: utf-8)

        Raises:
            PermissionError: If file cannot be written
        """
        ...

    def read_bytes(self, path: str) -> bytes:
        """Read bytes from a file.

        Args:
            path: File path to read

        Returns:
            File contents as bytes

        Raises:
            FileNotFoundError: If file does not exist
            PermissionError: If file cannot be read
        """
        ...

    def write_bytes(self, path: str, content: bytes) -> None:
        """Write bytes to a file.

        Args:
            path: File path to write
            content: Bytes content to write
```

```python
        Raises:
            PermissionError: If file cannot be written
        """
        ...

    def exists(self, path: str) -> bool:
        """Check if a file or directory exists.

        Args:
            path: Path to check

        Returns:
            True if path exists, False otherwise
        """
        ...

    def mkdir(self, path: str, parents: bool = False, exist_ok: bool = False) -> None:
        """Create a directory.

        Args:
            path: Directory path to create
            parents: Create parent directories if needed
            exist_ok: Don't raise error if directory exists

        Raises:
            FileExistsError: If directory exists and exist_ok is False
        """
        ...

class JsonPort(Protocol):
    """Port for JSON serialization/deserialization.

    Separates JSON operations from file I/O for better composability.
    """

    def loads(self, text: str) -> Any:
        """Parse JSON from string.

        Args:
            text: JSON string

        Returns:
            Parsed Python object

        Raises:
            ValueError: If JSON is invalid
        """
        ...

    def dumps(self, obj: Any, indent: int | None = None) -> str:
        """Serialize object to JSON string.

        Args:
            obj: Python object to serialize
            indent: Indentation spaces (None for compact)

        Returns:
            JSON string

        Raises:
            TypeError: If object is not serializable
        """
        ...

class EnvPort(Protocol):
    """Port for environment variable access.

    Allows core modules to access configuration without direct os.environ coupling.
```

```python
    """

    def get(self, key: str, default: str | None = None) -> str | None:
        """Get environment variable.

        Args:
            key: Environment variable name
            default: Default value if not set

        Returns:
            Environment variable value or default
        """
        ...

    def get_required(self, key: str) -> str:
        """Get required environment variable.

        Args:
            key: Environment variable name

        Returns:
            Environment variable value

        Raises:
            ValueError: If environment variable is not set
        """
        ...

    def get_bool(self, key: str, default: bool = False) -> bool:
        """Get environment variable as boolean.

        Args:
            key: Environment variable name
            default: Default value if not set

        Returns:
            Boolean value (true/false/yes/no/1/0)
        """
        ...

class ClockPort(Protocol):
    """Port for time operations.

    Allows core modules to get current time without direct datetime.now() calls.
    Enables time manipulation in tests.
    """

    def now(self) -> datetime:
        """Get current datetime.

        Returns:
            Current datetime
        """
        ...

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

        Returns:
            Current UTC datetime
        """
        ...

class LogPort(Protocol):
    """Port for logging operations.

    Allows core modules to log without coupling to specific logging framework.
    """
```

```
    def debug(self, message: str, **kwargs: Any) -> None:
        """Log debug message."""
        ...

    def info(self, message: str, **kwargs: Any) -> None:
        """Log info message."""
        ...

    def warning(self, message: str, **kwargs: Any) -> None:
        """Log warning message."""
        ...

    def error(self, message: str, **kwargs: Any) -> None:
        """Log error message."""
        ...

class PortCPPIngest(Protocol):
    """Port for CPP (Canon Policy Package) ingestion.

    Ingests documents and produces Canon Policy Packages with complete provenance.
    """

    def ingest(self, input_uri: str) -> Any:
        """Ingest document from URI and produce Canon Policy Package.

        Args:
            input_uri: URI to document (file://, http://, etc.)

        Returns:
            CanonPolicyPackage with complete chunk graph and metadata

        Requires:
            - Valid input URI
            - Accessible document at URI

        Ensures:
            - chunk_graph is not None
            - policy_manifest is not None
            - provenance_completeness == 1.0
        """
        ...


class PortCPPAdapter(Protocol):
    """Port for CPP to PreprocessedDocument adaptation.

    Converts Canon Policy Package to orchestrator's PreprocessedDocument format.

    Note: CPP is the legacy name. Use PortSPCAdapter for new code.
    """

    def to_preprocessed_document(self, cpp: Any, document_id: str) -> Any:
        """Convert CPP to PreprocessedDocument.

        Args:
            cpp: Canon Policy Package from ingestion
            document_id: Unique document identifier

        Returns:
            PreprocessedDocument for orchestrator

        Requires:
            - cpp with valid chunk_graph
            - cpp.policy_manifest exists
            - document_id is non-empty

        Ensures:
```

```
        - sentence_metadata is not empty
        - resolution_index is consistent
        - provenance_completeness == 1.0
    """
    ...


class PortSPCAdapter(Protocol):
    """Port for SPC (Smart Policy Chunks) to PreprocessedDocument adaptation.

    Converts Smart Policy Chunks to orchestrator's PreprocessedDocument format.
    This is the preferred terminology for new code.
    """

    def to_preprocessed_document(self, spc: Any, document_id: str) -> Any:
        """Convert SPC to PreprocessedDocument.

        Args:
            spc: Smart Policy Chunks package from ingestion
            document_id: Unique document identifier

        Returns:
            PreprocessedDocument for orchestrator

        Requires:
            - spc with valid chunk_graph
            - spc.policy_manifest exists
            - document_id is non-empty

        Ensures:
            - sentence_metadata is not empty
            - resolution_index is consistent
            - provenance_completeness == 1.0
        """
        ...


class PortSignalsClient(Protocol):
    """Port for fetching strategic signals.

    Retrieves policy-aware signals from memory or HTTP sources.
    Sematics: None return = 304 Not Modified or circuit breaker open.
    """

    def fetch(self, policy_area: str) -> Any | None:
        """Fetch signals for policy area.

        Args:
            policy_area: Policy domain (fiscal, salud, ambiente, etc.)

        Returns:
            SignalPack if available, None if 304/breaker open

        Requires:
            - policy_area is valid PolicyArea literal

        Ensures:
            - If not None, returns valid SignalPack with version
            - None is justified (304 or breaker state)
        """
        ...


class PortSignalsRegistry(Protocol):
    """Port for signal registry with TTL and LRU.

    Manages in-memory cache of strategic signals with expiration.
    """
```

```
    def put(self, pack: Any) -> None:
        """Store signal pack in registry.

        Args:
            pack: SignalPack to store

        Requires:
            - pack is valid SignalPack
            - pack.version is present
        """
        ...

    def get(self, policy_area: str) -> dict[str, Any] | None:
        """Retrieve signals for policy area.

        Args:
            policy_area: Policy domain

        Returns:
            Signal data if cached and not expired, None otherwise
        """
        ...

    def fingerprint(self) -> str:
        """Compute registry fingerprint for drift detection.

        Returns:
            BLAKE3 hash of current registry state
        """
        ...


class PortArgRouter(Protocol):
    """Port for argument routing and validation.

    Routes method calls with strict parameter validation.
    """

    def route(
        self,
        class_name: str,
        method_name: str,
        payload: dict[str, Any]
    ) -> tuple[tuple[Any, ...], dict[str, Any]]:
        """Route method call to (args, kwargs).

        Args:
            class_name: Target class name
            method_name: Target method name
            payload: Input parameters

        Returns:
            Tuple of (args, kwargs) for method call

        Requires:
            - class_name exists in registry
            - method_name exists on class
            - method signature is known or has **kwargs

        Ensures:
            - No silent parameter drops
            - All required args present
            - No unexpected kwargs (unless **kwargs in signature)
        """
        ...
```

```
class PortExecutor(Protocol):
    """Port for executing methods with configuration.

    Executes methods with injected executor config and signals.
    """

    def run(self, prompt: str, overrides: Any | None = None) -> Any:
        """Execute with prompt and optional config overrides.

        Args:
            prompt: Execution prompt/input
            overrides: Optional ExecutorConfig overrides

        Returns:
            Result with metadata including used_signals

        Requires:
            - ExecutorConfig is injected
            - SignalRegistry is available

        Ensures:
            - Result includes used_signals metadata
            - Execution is deterministic if seed is set
        """
        ...


class PortAggregate(Protocol):
    """Port for aggregating enriched chunks.

    Aggregates processed chunks into PyArrow tables.
    """

    def aggregate(self, enriched_chunks: list[dict[str, Any]]) -> Any:
        """Aggregate enriched chunks to PyArrow table.

        Args:
            enriched_chunks: List of enriched chunk dictionaries

        Returns:
            PyArrow Table with aggregated data

        Requires:
            - enriched_chunks has required fields
            - All chunks have consistent schema

        Ensures:
            - Returns valid pa.Table
            - All required columns present
        """
        ...


class PortScore(Protocol):
    """Port for scoring features.

    Computes scores from feature tables with specified metrics.
    """

    def score(self, features: Any, metrics: list[str]) -> Any:
        """Score features using specified metrics.

        Args:
            features: PyArrow Table with features
            metrics: List of metric names to compute

        Returns:
            Polars DataFrame with scores
```

```
        Requires:
            - features is valid pa.Table
            - metrics are declared and implemented
            - Required columns present in features

        Ensures:
            - Returns valid pl.DataFrame
            - All requested metrics computed
        """
        ...


class PortReport(Protocol):
```