```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 8
================================================================================
Generated: 2025-12-07T06:17:18.226843
Files in this batch: 17
================================================================================


================================================================================
FILE: src/farfan_pipeline/concurrency/concurrency.py
================================================================================

"""
Concurrency Module - Deterministic Worker Pool for Parallel Execution.

This module implements a deterministic WorkerPool for executing tasks in parallel
with the following features:
- Controlled max_workers for resource management
- Exponential backoff for retries
- Abortability for canceling pending tasks
- Per-task instrumentation and logging
- No race conditions or unwanted variability

Preconditions:
- Tasks and workers are declared before execution
- Each task is idempotent and thread-safe

Invariants:
- No interference between workers
- Deterministic task execution order within priority groups
- Thread-safe state management

Postconditions:
- Pool is usable by orchestrator/choreographer
- All resources are properly cleaned up
- No race conditions or variability in results
"""

from __future__ import annotations

import logging
import threading
import time
from concurrent.futures import Future, ThreadPoolExecutor, as_completed
from dataclasses import dataclass
from enum import Enum
from typing import TYPE_CHECKING, Any
from uuid import uuid4

if TYPE_CHECKING:
    from collections.abc import Callable

logger = logging.getLogger(__name__)

class TaskStatus(Enum):
    """Task execution status."""
    PENDING = "pending"
    RUNNING = "running"
    COMPLETED = "completed"
    FAILED = "failed"
    CANCELLED = "cancelled"
    RETRYING = "retrying"

class TaskExecutionError(Exception):
    """Exception raised when task execution fails."""
    pass

@dataclass
class WorkerPoolConfig:
    """Configuration for WorkerPool.

    Attributes:
        max_workers: Maximum number of concurrent workers (default: 50)
        task_timeout_seconds: Timeout for individual task execution (default: 180)
        max_retries: Maximum number of retries per task (default: 3)
        backoff_base_seconds: Base delay for exponential backoff (default: 1.0)
        backoff_max_seconds: Maximum backoff delay (default: 60.0)
        enable_instrumentation: Enable detailed logging and metrics (default: True)
    """
    max_workers: int = 50
    task_timeout_seconds: float = 180.0
    max_retries: int = 3
    backoff_base_seconds: float = 1.0
    backoff_max_seconds: float = 60.0
    enable_instrumentation: bool = True

@dataclass
class TaskMetrics:
    """Metrics for a single task execution.

    Attributes:
        task_id: Unique task identifier
        task_name: Human-readable task name
        status: Current task status
        start_time: Task start time (epoch seconds)
        end_time: Task end time (epoch seconds, None if not finished)
        execution_time_ms: Total execution time in milliseconds
        retries_used: Number of retries performed
        worker_id: ID of worker that executed the task
        error_message: Error message if task failed
    """
    task_id: str
    task_name: str
    status: TaskStatus
    start_time: float
    end_time: float | None = None
    execution_time_ms: float = 0.0
    retries_used: int = 0
    worker_id: str | None = None
    error_message: str | None = None

@dataclass
```

```
class TaskResult:
    """Result of a task execution.

    Attributes:
        task_id: Unique task identifier
        task_name: Human-readable task name
        success: Whether task succeeded
        result: Task result data (None if failed)
        error: Exception if task failed (None if succeeded)
        metrics: Execution metrics
    """
    task_id: str
    task_name: str
    success: bool
    result: Any = None
    error: Exception | None = None
    metrics: TaskMetrics | None = None

class WorkerPool:
    """
    Deterministic WorkerPool for parallel task execution.

    This pool provides controlled concurrency with the following guarantees:
    - No race conditions through thread-safe state management
    - Deterministic execution within priority groups
    - Proper resource cleanup and abort handling
    - Per-task instrumentation and logging

    Example:
        >>> config = WorkerPoolConfig(max_workers=10, max_retries=2)
        >>> pool = WorkerPool(config)
        >>>
        >>> def my_task(x):
        ...     return x * 2
        >>>
        >>> task_id = pool.submit_task("double_5", my_task, args=(5,))
        >>> results = pool.wait_for_all()
        >>> pool.shutdown()
    """

    def __init__(self, config: WorkerPoolConfig | None = None) -> None:
        """
        Initialize WorkerPool.

        Args:
            config: Pool configuration (uses defaults if None)
        """
        self.config = config or WorkerPoolConfig()
        self._executor: ThreadPoolExecutor | None = None
        self._futures: dict[str, Future] = {}
        self._task_info: dict[str, tuple[str, Callable, tuple, dict]] = {}
        self._metrics: dict[str, TaskMetrics] = {}
        self._lock = threading.Lock()
        self._abort_requested = threading.Event()
        self._is_shutdown = False

        logger.info(
            f"WorkerPool initialized: max_workers={self.config.max_workers}, "
            f"max_retries={self.config.max_retries}, "
            f"task_timeout={self.config.task_timeout_seconds}s"
        )

    def _create_executor(self) -> ThreadPoolExecutor:
        """Create thread pool executor lazily."""
        if self._executor is None:
            self._executor = ThreadPoolExecutor(
                max_workers=self.config.max_workers,
                thread_name_prefix="WorkerPool"
            )
        return self._executor

    def _calculate_backoff_delay(self, retry_count: int) -> float:
        """
        Calculate exponential backoff delay.

        Args:
            retry_count: Number of retries already attempted

        Returns:
            Delay in seconds, capped at backoff_max_seconds
        """
        delay = self.config.backoff_base_seconds * (2 ** retry_count)
        return min(delay, self.config.backoff_max_seconds)

    def _execute_task_with_retry(
        self,
        task_id: str,
        task_name: str,
        task_fn: Callable,
        args: tuple,
        kwargs: dict,
    ) -> Any:
        """
        Execute task with retry logic and exponential backoff.

        Args:
            task_id: Unique task identifier
            task_name: Human-readable task name
            task_fn: Task function to execute
            args: Positional arguments for task_fn
            kwargs: Keyword arguments for task_fn

        Returns:
            Task result

        Raises:
            TaskExecutionError: If task fails after all retries
        """
        worker_id = threading.current_thread().name
        retry_count = 0
        last_error = None

        # Initialize metrics
        with self._lock:
            self._metrics[task_id] = TaskMetrics(
                task_id=task_id,
                task_name=task_name,
                status=TaskStatus.RUNNING,
                start_time=time.time(),
                worker_id=worker_id
            )

        if self.config.enable_instrumentation:
            logger.info(f"[{task_id}] Starting task ’{task_name}’ on worker {worker_id}")

        while retry_count <= self.config.max_retries:
            # Check if abort was requested
            if self._abort_requested.is_set():
                with self._lock:
                    self._metrics[task_id].status = TaskStatus.CANCELLED
                    self._metrics[task_id].end_time = time.time()
                    self._metrics[task_id].execution_time_ms = (
                        (self._metrics[task_id].end_time - self._metrics[task_id].start_time) * 1000
                    )

                if self.config.enable_instrumentation:
                    logger.warning(f"[{task_id}] Task ’{task_name}’ cancelled due to abort request")

                raise TaskExecutionError(f"Task {task_name} cancelled due to abort request")

            try:
                # Execute task
                task_start = time.time()
                result = task_fn(*args, **kwargs)
                task_duration = (time.time() - task_start) * 1000

                # Update metrics on success
                with self._lock:
                    self._metrics[task_id].status = TaskStatus.COMPLETED
                    self._metrics[task_id].end_time = time.time()
                    self._metrics[task_id].execution_time_ms = task_duration
                    self._metrics[task_id].retries_used = retry_count

                if self.config.enable_instrumentation:
                    logger.info(
                        f"[{task_id}] Task ’{task_name}’ completed successfully "
                        f"in {task_duration:.2f}ms (retries: {retry_count})"
                    )

                return result

            except Exception as e:
                last_error = e

                # Update metrics on failure
                with self._lock:
                    self._metrics[task_id].retries_used = retry_count
                    self._metrics[task_id].error_message = str(e)

                if retry_count < self.config.max_retries:
                    # Calculate backoff delay
                    backoff_delay = self._calculate_backoff_delay(retry_count)

                    with self._lock:
                        self._metrics[task_id].status = TaskStatus.RETRYING

                    if self.config.enable_instrumentation:
                        logger.warning(
                            f"[{task_id}] Task '{task_name}' failed (attempt {retry_count + 1}), "
                            f"retrying after {backoff_delay:.2f}s: {e}"
                        )

                    # Wait before retrying (check abort periodically)
                    time.sleep(backoff_delay)
                    retry_count += 1
                else:
                    # All retries exhausted
                    with self._lock:
                        self._metrics[task_id].status = TaskStatus.FAILED
                        self._metrics[task_id].end_time = time.time()
                        self._metrics[task_id].execution_time_ms = (
                            (self._metrics[task_id].end_time - self._metrics[task_id].start_time) * 1000
                        )

                    if self.config.enable_instrumentation:
                        logger.error(
                            f"[{task_id}] Task '{task_name}' failed after {retry_count} retries: {e}"
                        )

                    raise TaskExecutionError(
                        f"Task {task_name} failed after {retry_count} retries: {last_error}"
                    ) from last_error

        # Should not reach here, but just in case
        raise TaskExecutionError(f"Task {task_name} failed: {last_error}")

    def submit_task(
        self,
        task_name: str,
        task_fn: Callable,
        args: tuple = (),
        kwargs: dict[str, Any] | None = None,
    ) -> str:
        """
        Submit a task for execution.

        Args:
            task_name: Human-readable task name for logging
            task_fn: Callable to execute
            args: Positional arguments for task_fn
            kwargs: Keyword arguments for task_fn

        Returns:
            Unique task identifier

        Raises:
            RuntimeError: If pool is shutdown
        """
        if self._is_shutdown:
            raise RuntimeError("Cannot submit tasks to a shutdown WorkerPool")

        kwargs = kwargs or {}
        task_id = str(uuid4())

        with self._lock:
            # Store task info for potential retries
            self._task_info[task_id] = (task_name, task_fn, args, kwargs)

            # Submit task to executor
            executor = self._create_executor()
            future = executor.submit(
                self._execute_task_with_retry,
                task_id,
                task_name,
                task_fn,
                args,
                kwargs
            )
            self._futures[task_id] = future

        if self.config.enable_instrumentation:
            logger.debug(f"[{task_id}] Task '{task_name}' submitted to pool")

        return task_id

    def get_task_result(self, task_id: str, timeout: float | None = None) -> TaskResult:
        """
        Get result of a specific task.

        Args:
            task_id: Task identifier returned by submit_task
            timeout: Maximum time to wait for result in seconds (None = wait forever)

        Returns:
            TaskResult with execution metrics

        Raises:
            KeyError: If task_id is not found
            TimeoutError: If timeout is exceeded
        """
        with self._lock:
            if task_id not in self._futures:
                raise KeyError(f"Task {task_id} not found")

            future = self._futures[task_id]
            task_name = self._task_info[task_id][0]

        try:
            timeout_to_use = timeout or self.config.task_timeout_seconds
            result = future.result(timeout=timeout_to_use)

            with self._lock:
                metrics = self._metrics.get(task_id)

            return TaskResult(
                task_id=task_id,
                task_name=task_name,
                success=True,
                result=result,
                metrics=metrics
            )

        except TimeoutError as e:
            with self._lock:
                metrics = self._metrics.get(task_id)
                if metrics:
                    metrics.status = TaskStatus.FAILED
                    metrics.error_message = f"Timeout after {timeout_to_use}s"

            return TaskResult(
                task_id=task_id,
                task_name=task_name,
                success=False,
                error=e,
                metrics=metrics
            )

        except Exception as e:
            with self._lock:
                metrics = self._metrics.get(task_id)

            return TaskResult(
                task_id=task_id,
                task_name=task_name,
                success=False,
                error=e,
                metrics=metrics
            )

    def wait_for_all(
        self,
        timeout: float | None = None,
        return_when: str = "ALL_COMPLETED"
    ) -> list[TaskResult]:
        """
        Wait for all submitted tasks to complete.

        Args:
            timeout: Maximum time to wait in seconds (None = wait forever)
            return_when: When to return - "ALL_COMPLETED" or "FIRST_EXCEPTION"

        Returns:
            List of TaskResults for all tasks

        Raises:
            TimeoutError: If timeout is exceeded before all tasks complete
        """
        if self.config.enable_instrumentation:
            logger.info(f"Waiting for {len(self._futures)} tasks to complete...")

        start_time = time.time()
        results = []

        with self._lock:
            all_futures = list(self._futures.items())

        try:
            # Use as_completed for better progress tracking
            completed_count = 0
            for future in as_completed(
                [f for _, f in all_futures],
                timeout=timeout
            ):
                completed_count += 1

                # Find task_id for this future
                task_id = None
                with self._lock:
                    for tid, f in all_futures:
                        if f == future:
                            task_id = tid
                            break

                if task_id:
                    result = self.get_task_result(task_id, timeout=0.1)
                    results.append(result)

                    if self.config.enable_instrumentation and completed_count % 10 == 0:
                        elapsed = time.time() - start_time
                        logger.info(
                            f"Progress: {completed_count}/{len(all_futures)} tasks completed "
                            f"({elapsed:.2f}s elapsed)"
                        )

                    # Check if we should return early on first exception
                    if return_when == "FIRST_EXCEPTION" and not result.success:
                        if self.config.enable_instrumentation:
                            logger.warning(
                                f"Returning early due to task failure: {result.task_name}"
                            )
                        break

            elapsed = time.time() - start_time
            if self.config.enable_instrumentation:
                successful = sum(1 for r in results if r.success)
                failed = sum(1 for r in results if not r.success)
                logger.info(
                    f"All tasks completed: {successful} succeeded, {failed} failed "
                    f"({elapsed:.2f}s total)"
                )

            return results

        except TimeoutError:
            elapsed = time.time() - start_time
            completed = len(results)
            pending = len(all_futures) - completed

            logger.error(
                f"Timeout after {elapsed:.2f}s: {completed} completed, {pending} pending"
            )

            # Get results for completed tasks
            for task_id, future in all_futures:
                if future.done() and task_id not in [r.task_id for r in results]:
                    try:
                        results.append(self.get_task_result(task_id, timeout=0.1))
                    except Exception as e:
                        logger.exception(f"Failed to get result for completed task {task_id}: {e}")

            raise TimeoutError(
                f"Timeout waiting for tasks: {completed}/{len(all_futures)} completed"
            )

    def abort_pending_tasks(self) -> int:
        """
        Request abort of all pending tasks.

        This sets the abort flag, which will be checked by running tasks
        at their next safe point (before retry or next iteration).

        Returns:
            Number of tasks that were still pending
        """
        self._abort_requested.set()

        pending_count = 0
        with self._lock:
            for task_id, future in self._futures.items():
                if not future.done():
                    future.cancel()
                    pending_count += 1

                    # Update metrics
                    if task_id in self._metrics:
                        self._metrics[task_id].status = TaskStatus.CANCELLED

        if self.config.enable_instrumentation:
            logger.warning(f"Abort requested: {pending_count} tasks cancelled")

        return pending_count

    def get_metrics(self) -> dict[str, TaskMetrics]:
        """
        Get execution metrics for all tasks.

        Returns:
            Dictionary mapping task_id to TaskMetrics
        """
        with self._lock:
            return dict(self._metrics)

    def get_summary_metrics(self) -> dict[str, Any]:
        """
        Get summary metrics for the pool.

        Returns:
            Dictionary with aggregated metrics
        """
        with self._lock:
            metrics_list = list(self._metrics.values())

        if not metrics_list:
            return {
                "total_tasks": 0,
                "completed": 0,
                "failed": 0,
                "cancelled": 0,
                "running": 0,
                "pending": 0,
                "avg_execution_time_ms": 0.0,
                "total_retries": 0,
            }

        completed = sum(1 for m in metrics_list if m.status == TaskStatus.COMPLETED)
        failed = sum(1 for m in metrics_list if m.status == TaskStatus.FAILED)
        cancelled = sum(1 for m in metrics_list if m.status == TaskStatus.CANCELLED)
        running = sum(1 for m in metrics_list if m.status == TaskStatus.RUNNING)
        pending = sum(1 for m in metrics_list if m.status == TaskStatus.PENDING)

        completed_tasks = [m for m in metrics_list if m.status == TaskStatus.COMPLETED]
        avg_time = (
            sum(m.execution_time_ms for m in completed_tasks) / len(completed_tasks)
            if completed_tasks else 0.0
        )

        total_retries = sum(m.retries_used for m in metrics_list)

        return {
            "total_tasks": len(metrics_list),
            "completed": completed,
            "failed": failed,
            "cancelled": cancelled,
            "running": running,
            "pending": pending,
            "avg_execution_time_ms": avg_time,
            "total_retries": total_retries,
        }

    def shutdown(self, wait: bool = True, cancel_futures: bool = False) -> None:
        """
        Shutdown the worker pool.

        Args:
            wait: If True, wait for all tasks to complete before shutdown
            cancel_futures: If True, cancel all pending tasks
        """
        if self._is_shutdown:
            return

        if cancel_futures:
            self.abort_pending_tasks()

        if self._executor is not None:
            if self.config.enable_instrumentation:
                logger.info(f"Shutting down WorkerPool (wait={wait})")

            self._executor.shutdown(wait=wait, cancel_futures=cancel_futures)
            self._executor = None

        self._is_shutdown = True

        if self.config.enable_instrumentation:
            summary = self.get_summary_metrics()
            logger.info(
                f"WorkerPool shutdown complete. "
                f"Completed: {summary['completed']}, "
                f"Failed: {summary['failed']}, "
                f"Cancelled: {summary['cancelled']}"
            )

    def __enter__(self):
        """Context manager entry."""
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        """Context manager exit."""
        self.shutdown(wait=True)
        return False



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




================================================================================
FILE: src/farfan_pipeline/config/dependency_lockdown.py
================================================================================

"""
Dependency Lockdown

Explicit allowlist of third-party modules permitted in F.A.R.F.A.N pipeline.

This is the SINGLE SOURCE OF TRUTH for:
- Which third-party packages are allowed to be imported.
- Which dynamic imports (if any) are whitelisted.

Maximum hardness interpretation:
- If a module is not in ALLOWED_THIRD_PARTY_MODULES, it MUST NOT be imported.
- If this file is missing or ALLOWED_THIRD_PARTY_MODULES is empty, policy_builder MUST fail hard.
- No silent fallback to requirements.txt is permitted.
"""

from typing import FrozenSet

# Third-party modules explicitly allowed for import
# These MUST correspond to packages in requirements.txt but are explicitly vetted
ALLOWED_THIRD_PARTY_MODULES: FrozenSet[str] = frozenset(
    {
        # Core data science
        "numpy",
        "np",
        "pandas",
        "pd",
        "scipy",
        "sklearn",
        "scikit-learn",
        "scikit_learn",
        # Deep learning
        "torch",
        "pytorch",
        "tensorflow",
        "tf",
        "keras",
        "tf_keras",
        # NLP / Transformers
        "transformers",
        "sentence_transformers",
        "sentencepiece",
        "tokenizers",
        # Vector DB / RAG
        "chromadb",
        "chroma",
        "faiss",
        # LLM integrations
        "langchain",
        "langchain_core",
        "langchain_community",
        "openai",
        "anthropic",
        # Web frameworks
        "pydantic",
        "pydantic_core",
        "fastapi",
        "uvicorn",
        "starlette",
        # Testing
        "pytest",
        "hypothesis",
        "mock",
        # Utilities
        "click",
        "tqdm",
        "requests",
        "httpx",
        "aiohttp",
        "psutil",
        "pypdf",
        "pypdf2",
        "pdfplumber",
        "reportlab",
        # Data formats
        "yaml",
        "pyyaml",
        "toml",
        "tomli",
        "msgpack",
        # Serialization
        "orjson",
        "ujson",
        # Async
        "asyncio",
        "aiofiles",
        # Environment
        "dotenv",
        "python-dotenv",
        "python_dotenv",
        # Logging / Observability
        "loguru",
        "structlog",
        # Type checking (dev)
        "typing_extensions",
        "mypy",
        "mypy_extensions",
        # Linting (dev)
        "ruff",
        "black",
        "isort",
        # Documentation (dev)
        "sphinx",
        "mkdocs",
        # Packaging
        "setuptools",
        "wheel",
        "pip",
        "pipdeptree",
        # Deprecated warnings for migration
        "deprecated",
        "warnings",
    }
)

# Dynamic imports explicitly allowed (optional, default empty)
# These are module names that can be imported via importlib.import_module()
ALLOWED_DYNAMIC_IMPORTS: FrozenSet[str] = frozenset(
    {
        # Add dynamic imports here if needed
        # Example: "farfan_core.plugins.optional_module"
    }
)



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

"""
Centralized Path Configuration for F.A.R.F.A.N
==============================================

This module provides a single source of truth for all filesystem paths
used throughout the project. This ensures:

1. Portability: Works in development and production
2. Configurability: Paths can be overridden via environment variables
3. Consistency: All modules use the same path definitions
4. Testability: Paths can be mocked for testing

Author: Python Pipeline Expert
Date: 2025-11-15

Usage:
    from farfan_core.config.paths import DATA_DIR, OUTPUT_DIR, CACHE_DIR

    questionnaire = DATA_DIR / 'questionnaire_monolith.json'
    report = OUTPUT_DIR / 'analysis_report.json'
"""

import logging
import os
import sys
from pathlib import Path
from typing import Final, Tuple

# ============================================================================
# Project Root Detection
# ============================================================================

logger = logging.getLogger("farfan_core.config.paths")


def _detect_project_root() -> Tuple[Path, str]:
    """
    Detect project root directory reliably in both dev and production.

    Strategy:
    1. If running from installed package: Use site-packages parent
    2. If running from source: Navigate from this file to project root
    3. Fallback: Use environment variable SAAAAAA_PROJECT_ROOT
    """
    # Check environment variable first (explicit override)
    if env_root := os.getenv('SAAAAAA_PROJECT_ROOT'):
        return Path(env_root).resolve(), "env"

    # Detect from this file’s location
    # src/farfan_core/config/paths.py â\206\222 project_root
    this_file = Path(__file__).resolve()

    # Navigate up: paths.py -> config -> farfan_core -> src -> project_root
    candidate = this_file.parents[3]

    # Verify this looks like our project (has setup.py or pyproject.toml)
    if (candidate / ’setup.py’).exists() or (candidate / ’pyproject.toml’).exists():
        return candidate, "markers"

    raise RuntimeError(
        "Unable to determine project root. "
        "Set the SAAAAAA_PROJECT_ROOT environment variable."
    )


# Project root (base for all other paths)
PROJECT_ROOT, PROJECT_ROOT_SOURCE = _detect_project_root()
logger.info("Project root detected via %s: %s", PROJECT_ROOT_SOURCE, PROJECT_ROOT)

# ============================================================================
# Core Directories
# ============================================================================

# Source code directory
SRC_DIR: Final[Path] = PROJECT_ROOT / ’src’ / ’farfan_core’

# Package root (for importlib.resources)
PACKAGE_ROOT: Final[Path] = SRC_DIR

# ============================================================================
# Data Directories (Configurable)
# ============================================================================

# Input data directory
DATA_DIR: Final[Path] = Path(
    os.getenv(’SAAAAAA_DATA_DIR’, str(PROJECT_ROOT / ’data’))
).resolve()

# Output directory for generated reports
OUTPUT_DIR: Final[Path] = Path(
    os.getenv(’SAAAAAA_OUTPUT_DIR’, str(PROJECT_ROOT / ’output’))
).resolve()

# Cache directory for temporary artifacts
CACHE_DIR: Final[Path] = Path(
    os.getenv(’SAAAAAA_CACHE_DIR’, str(PROJECT_ROOT / ’.cache’))
).resolve()

# Logs directory
LOGS_DIR: Final[Path] = Path(
    os.getenv(’SAAAAAA_LOGS_DIR’, str(PROJECT_ROOT / ’logs’))
).resolve()
```

```

# ============================================================================
# Configuration Directories
# ============================================================================

# Configuration files directory
CONFIG_DIR: Final[Path] = SRC_DIR / 'config'

# Rules and schemas directory
RULES_DIR: Final[Path] = PROJECT_ROOT / 'config' / 'rules'
SCHEMAS_DIR: Final[Path] = PROJECT_ROOT / 'config' / 'schemas'

# ============================================================================
# Common Data Files
# ============================================================================

# Questionnaire monolith (canonical)
QUESTIONNAIRE_FILE: Final[Path] = DATA_DIR / 'questionnaire_monolith.json'

# Method catalog
METHOD_CATALOG_FILE: Final[Path] = DATA_DIR / 'metodos' / 'catalogo_metodos.json'

# ============================================================================
# Test Directories
# ============================================================================

# Test data directory (fixtures, golden files, etc.)
TEST_DATA_DIR: Final[Path] = PROJECT_ROOT / 'tests' / 'data'

# Test output directory (temporary outputs from tests)
TEST_OUTPUT_DIR: Final[Path] = PROJECT_ROOT / 'tests' / 'output'

# ============================================================================
# Utilities
# ============================================================================

def ensure_directories_exist() -> None:
    """
    Create all required directories if they don't exist.

    This should be called at application startup to ensure the
    filesystem is properly initialized.
    """
    required_dirs = [
        DATA_DIR,
        OUTPUT_DIR,
        CACHE_DIR,
        LOGS_DIR,
        TEST_OUTPUT_DIR,
    ]

    for dir_path in required_dirs:
        dir_path.mkdir(parents=True, exist_ok=True)


def get_output_path(plan_name: str, suffix: str = '') -> Path:
    """
    Get output path for a specific plan analysis.

    Args:
        plan_name: Name of the plan (e.g., "cpp_plan_1")
        suffix: Optional suffix for the output file

    Returns:
        Path to output file

    Example:
        >>> output_path = get_output_path("cpp_plan_1", "micro_analysis.json")
        >>> # Returns: output/cpp_plan_1/micro_analysis.json
    """
    plan_dir = OUTPUT_DIR / plan_name
    plan_dir.mkdir(parents=True, exist_ok=True)

    if suffix:
        return plan_dir / suffix
    return plan_dir


def get_cache_path(namespace: str, key: str) -> Path:
    """
    Get cache path for a specific namespace and key.

    Args:
        namespace: Cache namespace (e.g., "embeddings", "chunks")
        key: Cache key (will be sanitized)

    Returns:
        Path to cache file

    Example:
        >>> cache_path = get_cache_path("embeddings", "plan_123_chunk_5")
        >>> # Returns: .cache/embeddings/plan_123_chunk_5
    """
    namespace_dir = CACHE_DIR / namespace
    namespace_dir.mkdir(parents=True, exist_ok=True)

    # Sanitize key (remove dangerous characters)
    safe_key = key.replace(’/’, ’_’).replace(’\\’, ’_’).replace(’..’, ’_’)

    return namespace_dir / safe_key


def validate_paths() -> bool:
    """
    Validate that all critical paths exist and are accessible.

    Returns:
        True if all paths are valid, False otherwise
    """
    issues = []

    # Check PROJECT_ROOT
    if not PROJECT_ROOT.exists():
        issues.append(f"PROJECT_ROOT does not exist: {PROJECT_ROOT}")

    # Check SRC_DIR
    if not SRC_DIR.exists():
        issues.append(f"SRC_DIR does not exist: {SRC_DIR}")

    # Check critical data files
    if not QUESTIONNAIRE_FILE.exists():
        issues.append(f"Questionnaire file not found: {QUESTIONNAIRE_FILE}")

    if issues:
        print("â\232 ï¸\217  Path validation issues:", file=sys.stderr)
        for issue in issues:
            print(f"   - {issue}", file=sys.stderr)
        return False

    return True


# ============================================================================
# Initialization
# ============================================================================

# Ensure directories exist on import (safe, idempotent)
ensure_directories_exist()

# ============================================================================
# Compatibility Shims (DEPRECATED - for migration period)
# ============================================================================

# These provide backward compatibility during migration
# TODO: Remove these after migration is complete

def proj_root() -> Path:
    """DEPRECATED: Use PROJECT_ROOT instead."""
    import warnings
    warnings.warn(
        "proj_root() is deprecated. Use PROJECT_ROOT constant instead.",
        DeprecationWarning,
        stacklevel=2
    )
    return PROJECT_ROOT


def reports_dir() -> Path:
    """DEPRECATED: Use OUTPUT_DIR instead."""
    import warnings
    warnings.warn(
        "reports_dir() is deprecated. Use OUTPUT_DIR constant instead.",
        DeprecationWarning,
        stacklevel=2
    )
    return OUTPUT_DIR


```

```
# ============================================================================
# Debug Information
# ============================================================================

if __name__ == "__main__":
    """Print path configuration for debugging."""
    print("=" * 80)
    print("F.A.R.F.A.N Path Configuration")
    print("=" * 80)
    print()
    print(f"PROJECT_ROOT:     {PROJECT_ROOT}")
    print(f"SRC_DIR:          {SRC_DIR}")
    print(f"DATA_DIR:         {DATA_DIR}")
    print(f"OUTPUT_DIR:       {OUTPUT_DIR}")
    print(f"CACHE_DIR:        {CACHE_DIR}")
    print(f"LOGS_DIR:         {LOGS_DIR}")
    print()
    print(f"QUESTIONNAIRE:    {QUESTIONNAIRE_FILE}")
    print(f"  Exists: {QUESTIONNAIRE_FILE.exists()}")
    print()
    print("Validation:", "â\234\205 PASS" if validate_paths() else "â\235\214 FAIL")
    print()



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

"""
Contracts Package
"""
# Expose contracts for easier import
from farfan_pipeline.contracts.routing_contract import RoutingContract
from farfan_pipeline.contracts.snapshot_contract import SnapshotContract
from farfan_pipeline.contracts.context_immutability import ContextImmutabilityContract
from farfan_pipeline.contracts.permutation_invariance import PermutationInvarianceContract
from farfan_pipeline.contracts.budget_monotonicity import BudgetMonotonicityContract
from farfan_pipeline.contracts.total_ordering import TotalOrderingContract
from farfan_pipeline.contracts.retriever_contract import RetrieverContract
from farfan_pipeline.contracts.alignment_stability import AlignmentStabilityContract
from farfan_pipeline.contracts.idempotency_dedup import IdempotencyContract
from farfan_pipeline.contracts.risk_certificate import RiskCertificateContract
from farfan_pipeline.contracts.monotone_compliance import MonotoneComplianceContract
from farfan_pipeline.contracts.failure_fallback import FailureFallbackContract
from farfan_pipeline.contracts.concurrency_determinism import ConcurrencyDeterminismContract
from farfan_pipeline.contracts.traceability import TraceabilityContract
from farfan_pipeline.contracts.refusal import RefusalContract



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

"""
```

```
Alignment Stability Contract (ASC) - Implementation
"""
import hashlib
import json
from typing import List, Dict, Any, Tuple

class AlignmentStabilityContract:
    @staticmethod
    def compute_alignment(
        sections: List[str],
        standards: List[str],
        params: Dict[str, Any]
    ) -> Dict[str, Any]:
        """
        Simulates Optimal Transport (EGW) alignment.
        In a real system, this would use POT (Python Optimal Transport).
        """
        # Deterministic simulation based on inputs
        input_str = f"{sections}:{standards}:{json.dumps(params, sort_keys=True)}"
        hasher = hashlib.blake2b(input_str.encode(), digest_size=32)
        digest = hasher.hexdigest()

        # Simulate a plan (matrix) digest
        plan_digest = hashlib.blake2b(f"plan_{digest}".encode()).hexdigest()

        # Simulate cost and unmatched mass
        cost = int(digest[:4], 16) / 1000.0
        unmatched_mass = int(digest[4:8], 16) / 10000.0

        return {
            "plan_digest": plan_digest,
            "cost": cost,
            "unmatched_mass": unmatched_mass
        }

    @staticmethod
    def verify_stability(
        sections: List[str],
        standards: List[str],
        params: Dict[str, Any]
    ) -> bool:
        """
        Verifies reproducibility with fixed hyperparameters.
        """
        res1 = AlignmentStabilityContract.compute_alignment(sections, standards, params)
        res2 = AlignmentStabilityContract.compute_alignment(sections, standards, params)
        return res1 == res2



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

"""
Budget & Monotonicity Contract (BMC) - Implementation
```

```
"""
from typing import List, Dict, Set

class BudgetMonotonicityContract:
    @staticmethod
    def solve_knapsack(items: Dict[str, float], budget: float) -> Set[str]:
        """
        Solves a knapsack-like problem (selecting items within budget).
        To ensure monotonicity (S*(B1) â\212\206 S*(B2)), we use a greedy approach based on cost/benefit
        or simply cost if benefit is uniform.
        Here we assume we want to maximize count of items, so we pick cheapest first.
        """
        sorted_items = sorted(items.items(), key=lambda x: x[1]) # Sort by cost ascending

        selected = set()
        current_cost = 0.0

        for item_id, cost in sorted_items:
            if current_cost + cost <= budget:
                selected.add(item_id)
                current_cost += cost
            else:
                break

        return selected

    @staticmethod
    def verify_monotonicity(items: Dict[str, float], budgets: List[float]) -> bool:
        """
        Verifies S*(B1) â\212\206 S*(B2) for B1 < B2.
        """
        sorted_budgets = sorted(budgets)
        prev_solution = None

        for b in sorted_budgets:
            solution = BudgetMonotonicityContract.solve_knapsack(items, b)
            if prev_solution is not None:
                if not prev_solution.issubset(solution):
                    return False
            prev_solution = solution

        return True



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

"""
Concurrency Determinism Contract (CDC) - Implementation
"""
import hashlib
import json
import threading
import time
```

```
from typing import List, Any, Callable

class ConcurrencyDeterminismContract:
    @staticmethod
    def execute_concurrently(
        func: Callable[[Any], Any],
        inputs: List[Any],
        workers: int
    ) -> List[Any]:
        """
        Simulates concurrent execution.
        To ensure determinism, results must be sorted or indexed by input ID.
        """
        results = [None] * len(inputs)

        def worker(idx, inp):
            # Simulate work
            time.sleep(0.001)
            results[idx] = func(inp)

        threads = []
        for i, inp in enumerate(inputs):
            t = threading.Thread(target=worker, args=(i, inp))
            threads.append(t)
            t.start()

            if len(threads) >= workers:
                for t in threads:
                    t.join()
                threads = []

        for t in threads:
            t.join()

        return results

    @staticmethod
    def verify_determinism(
        func: Callable[[Any], Any],
        inputs: List[Any]
    ) -> bool:
        """
        Verifies that 1 worker vs N workers produces hash-equal outputs.
        """
        res_seq = ConcurrencyDeterminismContract.execute_concurrently(func, inputs, workers=1)
        res_conc = ConcurrencyDeterminismContract.execute_concurrently(func, inputs, workers=4)

        hash1 = hashlib.blake2b(json.dumps(res_seq, sort_keys=True).encode()).hexdigest()
        hash2 = hashlib.blake2b(json.dumps(res_conc, sort_keys=True).encode()).hexdigest()

        return hash1 == hash2



================================================================================
FILE: src/farfan_pipeline/contracts/context_immutability.py
```

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

from __future__ import annotations

import json
from dataclasses import is_dataclass, fields, FrozenInstanceError
from typing import Any
from collections.abc import Mapping as ABCMapping
from types import MappingProxyType
from collections.abc import Sequence as ABCSequence  # optional, for type checks

class ContextImmutabilityContract:
    @staticmethod
    def _to_plain(obj: Any) -> Any:
        """
        Convert dataclasses + immutable containers (MappingProxyType, tuples, frozensets)
        into plain Python structures without deepcopy(). This avoids mappingproxy pickling.
        """
        if is_dataclass(obj):
            return {f.name: ContextImmutabilityContract._to_plain(getattr(obj, f.name)) for f in fields(obj)}
        if isinstance(obj, (MappingProxyType, ABCMapping)):
            return {k: ContextImmutabilityContract._to_plain(v) for k, v in obj.items()}
        if isinstance(obj, (tuple, list, set, frozenset)):
            return [ContextImmutabilityContract._to_plain(v) for v in obj]
        return obj

    @staticmethod
    def _to_canonical(obj: Any) -> Any:
        # Sort mapping keys deterministically; lists already deterministic after _to_plain
        if isinstance(obj, dict):
            return {k: ContextImmutabilityContract._to_canonical(obj[k]) for k in sorted(obj.keys())}
        if isinstance(obj, list):
            return [ContextImmutabilityContract._to_canonical(v) for v in obj]
        return obj

    @staticmethod
    def canonical_digest(ctx: Any) -> str:
        # Build plain JSON-safe object without deepcopy(), then canonicalize & hash.
        plain = ContextImmutabilityContract._to_plain(ctx)
        canon = ContextImmutabilityContract._to_canonical(plain)
        s = json.dumps(canon, sort_keys=True, ensure_ascii=False, separators=(",", ":"))
        try:
            import blake3
            return blake3.blake3(s.encode("utf-8")).hexdigest()
        except Exception:
            import hashlib
            return hashlib.sha256(s.encode("utf-8")).hexdigest()

    @staticmethod
    def verify_immutability(ctx: Any) -> str:
        """
        Attempt to mutate: (1) top-level attribute, (2) deep mapping.
        Both must fail. Return canonical digest for equality comparisons.
        """
        # 1) Top-level attribute mutation must fail
        try:
            setattr(ctx, "traceability_id", "MUTATE_ME")
            raise RuntimeError("Mutation succeeded but should have failed!")
        except (FrozenInstanceError, AttributeError, TypeError):
            pass  # expected

        # 2) Deep mapping mutation must fail
        deep_map = getattr(ctx, "dnp_standards", None)
        if isinstance(deep_map, (MappingProxyType, ABCMapping)):
            try:
                deep_map["__MUTATE__"] = 1  # type: ignore[index]
                raise RuntimeError("Deep mutation succeeded but should have failed!")
            except (TypeError, AttributeError):
                pass  # expected

        # 3) Deterministic canonical digest
        return ContextImmutabilityContract.canonical_digest(ctx)



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

"""
Failure & Fallback Contract (FFC) - Implementation
"""
from typing import Callable, Any, Dict, Type, Tuple

class FailureFallbackContract:
    @staticmethod
    def execute_with_fallback(
        func: Callable,
        fallback_value: Any,
        expected_exceptions: Tuple[Type[Exception], ...]
    ) -> Any:
        """
        Executes func. If it raises an expected exception, returns fallback_value.
        Ensures determinism and no side effects (simulated).
        """
        try:
            return func()
        except expected_exceptions:
            return fallback_value

    @staticmethod
    def verify_fallback_determinism(
        func: Callable,
        fallback_value: Any,
        exception_type: Type[Exception]
    ) -> bool:
        """
        Verifies that repeated failures produce identical fallback values.
        """
        res1 = FailureFallbackContract.execute_with_fallback(func, fallback_value, (exception_type,))
        res2 = FailureFallbackContract.execute_with_fallback(func, fallback_value, (exception_type,))
        return res1 == res2
```

```



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

"""
Contract Governance Utilities
"""
from typing import Type, Any, Callable
import functools

def uses_contract(contract_class: Type[Any]) -> Callable:
    """
    Decorator to explicitly declare that a function or class relies on a specific contract.
    This serves as documentation and allows for static analysis of contract dependencies.

    Usage:
        @uses_contract(RoutingContract)
        def my_function():
            ...
    """
    def decorator(obj: Any) -> Any:
        if not hasattr(obj, "_contract_dependencies"):
            obj._contract_dependencies = []
        obj._contract_dependencies.append(contract_class)

        # If it’s a function, wrap it to preserve metadata
        if callable(obj) and not isinstance(obj, type):
            @functools.wraps(obj)
            def wrapper(*args, **kwargs):
                return obj(*args, **kwargs)
            wrapper._contract_dependencies = obj._contract_dependencies
            return wrapper

        return obj
    return decorator



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

"""
Idempotency & De-dup Contract (IDC) - Implementation
"""
import hashlib
import json
from typing import List, Dict, Any, Set

class EvidenceStore:
    def __init__(self):
        self.evidence: Dict[str, Any] = {} # content_hash -> evidence
        self.duplicates_blocked = 0

    def add(self, item: Dict[str, Any]):
        # Calculate content hash
        content_hash = hashlib.blake2b(json.dumps(item, sort_keys=True).encode()).hexdigest()

        if content_hash in self.evidence:
            self.duplicates_blocked += 1
        else:
            self.evidence[content_hash] = item

    def state_hash(self) -> str:
        # Hash of sorted keys to ensure order independence
        sorted_keys = sorted(self.evidence.keys())
        return hashlib.blake2b(json.dumps(sorted_keys).encode()).hexdigest()

class IdempotencyContract:
    @staticmethod
    def verify_idempotency(items: List[Dict[str, Any]]) -> Dict[str, Any]:
        store = EvidenceStore()
        for item in items:
            store.add(item)

        return {
            "state_hash": store.state_hash(),
            "duplicates_blocked": store.duplicates_blocked,
            "count": len(store.evidence)
        }



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

"""
Monotone Compliance Contract (MCC) - Implementation
"""
from typing import Set, Dict, Any
from enum import IntEnum

class Label(IntEnum):
    UNSAT = 0
    PARTIAL = 1
    SAT = 2

class MonotoneComplianceContract:
    @staticmethod
    def evaluate(evidence: Set[str], rules: Dict[str, Any]) -> Label:
        """
        Evaluates label based on evidence and Horn-like clauses.
        Simple logic:
        - SAT if all ’sat_reqs’ present
        - PARTIAL if all ’partial_reqs’ present
        - UNSAT otherwise
        """
        sat_reqs = set(rules.get("sat_reqs", []))
        partial_reqs = set(rules.get("partial_reqs", []))

        if sat_reqs.issubset(evidence):
            return Label.SAT
        elif partial_reqs.issubset(evidence):
            return Label.PARTIAL
        else:
            return Label.UNSAT

    @staticmethod
    def verify_monotonicity(
        evidence_subset: Set[str],
        evidence_superset: Set[str],
        rules: Dict[str, Any]
    ) -> bool:
        """
        Verifies label(E’) >= label(E) for E â\212\206 E’.
        """
        if not evidence_subset.issubset(evidence_superset):
            raise ValueError("Subset is not contained in superset")

        l1 = MonotoneComplianceContract.evaluate(evidence_subset, rules)
        l2 = MonotoneComplianceContract.evaluate(evidence_superset, rules)

        return l2 >= l1



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

"""
Permutation-Invariance Contract (PIC) - Implementation
"""
import hashlib
from typing import List, Any, Callable

class PermutationInvarianceContract:
    @staticmethod
    def aggregate(items: List[Any], transform: Callable[[Any], float]) -> float:
        """
        Implements f(S) = Ï\225(Î£ Ï\210(x)) pattern for permutation invariance.
        Here, sum is the aggregation function (symmetric).
        """
        # Ï\210(x) = transform(x)
        transformed = [transform(x) for x in items]

        # Î£ Ï\210(x) - Sum is order-independent (within floating point limits, usually)
        # For strict bitwise invariance with floats, we might need to sort or use exact arithmetic.
        # But the requirement asks for "numerical tolerance".
        total = sum(transformed)

        # Ï\225(x) = identity (for this example)
        return total

    @staticmethod
    def verify_invariance(items: List[Any], transform: Callable[[Any], float]) -> str:
        """
        Calculates digest of the aggregation.
        """
        result = PermutationInvarianceContract.aggregate(items, transform)
        return hashlib.blake2b(str(result).encode()).hexdigest()



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

"""
Refusal Contract (RefC) - Implementation
"""
from typing import Dict, Any

class RefusalError(Exception):
    pass

class RefusalContract:
    @staticmethod
    def check_prerequisites(context: Dict[str, Any]):
        """
        Confirma que ante prerequisitos fallidos el sistema rehÃ°sa con motivo tipado.
        """
        if "mandatory" not in context:
            raise RefusalError("Missing mandatory field")

        if context.get("alpha", 1.0) > 0.5:
             raise RefusalError("Alpha violation")

        if "sigma" not in context:
            raise RefusalError("Sigma absent")

    @staticmethod
    def verify_refusal(context: Dict[str, Any]) -> str:
        try:
            RefusalContract.check_prerequisites(context)
            return "OK"
        except RefusalError as e:
            return str(e)



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

"""
Retriever Contract (ReC) - Implementation
"""
import hashlib
import json
```

```
from typing import List, Dict, Any

class RetrieverContract:
    @staticmethod
    def retrieve(query: str, filters: Dict[str, Any], index_hash: str, top_k: int = 5) -> List[Dict[str, Any]]:
        """
        Simulates hybrid retrieval (patterns+dimension+Ï\203+Î\230).
        In a real system, this would call FAISS/Pyserini.
        Here we simulate deterministic retrieval based on inputs.
        """
        # Deterministic simulation
        input_data = f"{query}:{json.dumps(filters, sort_keys=True)}:{index_hash}"
        hasher = hashlib.blake2b(input_data.encode(), digest_size=32)

        results = []
        current_hash = hasher.hexdigest()

        for i in range(top_k):
            doc_hash = hashlib.blake2b(f"{current_hash}:{i}".encode()).hexdigest()
            results.append({
                "id": f"doc_{doc_hash[:8]}",
                "score": 0.9 - (i * 0.1),
                "content_hash": doc_hash
            })

        return results

    @staticmethod
    def verify_determinism(query: str, filters: Dict[str, Any], index_hash: str) -> str:
        """
        Returns a digest of the top-K results to verify determinism.
        """
        results = RetrieverContract.retrieve(query, filters, index_hash)
        # De-dup by content_hash is implicit if retrieval is deterministic,
        # but we can enforce it here if needed.

        # Serialize results for hashing
        return hashlib.blake2b(json.dumps(results, sort_keys=True).encode()).hexdigest()



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

"""
Risk Certificate Contract (RCC) - Implementation
"""
import numpy as np
from typing import List, Tuple, Dict

class RiskCertificateContract:
    @staticmethod
    def conformal_prediction(
        calibration_scores: List[float],
        alpha: float
    ) -> float:
        """
        Computes the quantile for conformal prediction.
        q = (1 - alpha) * (n + 1) / n corrected quantile
        """
        n = len(calibration_scores)
        q_level = np.ceil((n + 1) * (1 - alpha)) / n
        q_level = min(1.0, max(0.0, q_level))

        # Use numpy quantile
        return np.quantile(calibration_scores, q_level, method='higher')

    @staticmethod
    def verify_risk(
        calibration_data: List[float],
        holdout_data: List[float],
        alpha: float,
        seed: int
    ) -> Dict[str, float]:
        """
        Verifies that empirical coverage is approx (1-alpha) and risk <= alpha.
        """
        np.random.seed(seed)

        # Compute threshold from calibration data
        threshold = RiskCertificateContract.conformal_prediction(calibration_data, alpha)

        # Check coverage on holdout
        covered = [s <= threshold for s in holdout_data]
        coverage = sum(covered) / len(holdout_data)
        risk = 1.0 - coverage

        return {
            "alpha": alpha,
            "threshold": float(threshold),
            "coverage": coverage,
            "risk": risk
        }


```