

```
1: =====
2: F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 8
3: =====
4: Generated: 2025-12-07T06:17:18.226843
5: Files in this batch: 17
6: =====
7:
8:
9: =====
10: FILE: src/farfan_pipeline/concurrency/concurrency.py
11: =====
12:
13: """
14: Concurrency Module - Deterministic Worker Pool for Parallel Execution.
15:
16: This module implements a deterministic WorkerPool for executing tasks in parallel
17: with the following features:
18: - Controlled max_workers for resource management
19: - Exponential backoff for retries
20: - Abortability for canceling pending tasks
21: - Per-task instrumentation and logging
22: - No race conditions or unwanted variability
23:
24: Preconditions:
25: - Tasks and workers are declared before execution
26: - Each task is idempotent and thread-safe
27:
28: Invariants:
29: - No interference between workers
30: - Deterministic task execution order within priority groups
31: - Thread-safe state management
32:
33: Postconditions:
34: - Pool is usable by orchestrator/choreographer
35: - All resources are properly cleaned up
36: - No race conditions or variability in results
37: """
38:
39: from __future__ import annotations
40:
41: import logging
42: import threading
43: import time
44: from concurrent.futures import Future, ThreadPoolExecutor, as_completed
45: from dataclasses import dataclass
46: from enum import Enum
47: from typing import TYPE_CHECKING, Any
48: from uuid import uuid4
49:
50: if TYPE_CHECKING:
51:     from collections.abc import Callable
52:
53: logger = logging.getLogger(__name__)
54:
55: class TaskStatus(Enum):
56:     """Task execution status."""
```

```
57:     PENDING = "pending"
58:     RUNNING = "running"
59:     COMPLETED = "completed"
60:     FAILED = "failed"
61:     CANCELLED = "cancelled"
62:     RETRYING = "retrying"
63:
64: class TaskExecutionError(Exception):
65:     """Exception raised when task execution fails."""
66:     pass
67:
68: @dataclass
69: class WorkerPoolConfig:
70:     """Configuration for WorkerPool.
71:
72:     Attributes:
73:         max_workers: Maximum number of concurrent workers (default: 50)
74:         task_timeout_seconds: Timeout for individual task execution (default: 180)
75:         max_retries: Maximum number of retries per task (default: 3)
76:         backoff_base_seconds: Base delay for exponential backoff (default: 1.0)
77:         backoff_max_seconds: Maximum backoff delay (default: 60.0)
78:         enable_instrumentation: Enable detailed logging and metrics (default: True)
79:     """
80:     max_workers: int = 50
81:     task_timeout_seconds: float = 180.0
82:     max_retries: int = 3
83:     backoff_base_seconds: float = 1.0
84:     backoff_max_seconds: float = 60.0
85:     enable_instrumentation: bool = True
86:
87: @dataclass
88: class TaskMetrics:
89:     """Metrics for a single task execution.
90:
91:     Attributes:
92:         task_id: Unique task identifier
93:         task_name: Human-readable task name
94:         status: Current task status
95:         start_time: Task start time (epoch seconds)
96:         end_time: Task end time (epoch seconds, None if not finished)
97:         execution_time_ms: Total execution time in milliseconds
98:         retries_used: Number of retries performed
99:         worker_id: ID of worker that executed the task
100:        error_message: Error message if task failed
101:    """
102:    task_id: str
103:    task_name: str
104:    status: TaskStatus
105:    start_time: float
106:    end_time: float | None = None
107:    execution_time_ms: float = 0.0
108:    retries_used: int = 0
109:    worker_id: str | None = None
110:    error_message: str | None = None
111:
112: @dataclass
```

```
113: class TaskResult:
114:     """Result of a task execution.
115:
116:     Attributes:
117:         task_id: Unique task identifier
118:         task_name: Human-readable task name
119:         success: Whether task succeeded
120:         result: Task result data (None if failed)
121:         error: Exception if task failed (None if succeeded)
122:         metrics: Execution metrics
123:     """
124:     task_id: str
125:     task_name: str
126:     success: bool
127:     result: Any = None
128:     error: Exception | None = None
129:     metrics: TaskMetrics | None = None
130:
131: class WorkerPool:
132:     """
133:     Deterministic WorkerPool for parallel task execution.
134:
135:     This pool provides controlled concurrency with the following guarantees:
136:     - No race conditions through thread-safe state management
137:     - Deterministic execution within priority groups
138:     - Proper resource cleanup and abort handling
139:     - Per-task instrumentation and logging
140:
141:     Example:
142:         >>> config = WorkerPoolConfig(max_workers=10, max_retries=2)
143:         >>> pool = WorkerPool(config)
144:         >>>
145:         >>> def my_task(x):
146:             ...     return x * 2
147:         >>>
148:         >>> task_id = pool.submit_task("double_5", my_task, args=(5,))
149:         >>> results = pool.wait_for_all()
150:         >>> pool.shutdown()
151:     """
152:
153:     def __init__(self, config: WorkerPoolConfig | None = None) -> None:
154:         """
155:             Initialize WorkerPool.
156:
157:             Args:
158:                 config: Pool configuration (uses defaults if None)
159:             """
160:             self.config = config or WorkerPoolConfig()
161:             self._executor: ThreadPoolExecutor | None = None
162:             self._futures: dict[str, Future] = {}
163:             self._task_info: dict[str, tuple[str, Callable, tuple, dict]] = {}
164:             self._metrics: dict[str, TaskMetrics] = {}
165:             self._lock = threading.Lock()
166:             self._abort_requested = threading.Event()
167:             self._is_shutdown = False
168:
```

```
169:         logger.info(
170:             f"WorkerPool initialized: max_workers={self.config.max_workers}, "
171:             f"max_retries={self.config.max_retries}, "
172:             f"task_timeout={self.config.task_timeout_seconds}s"
173:         )
174:
175:     def _create_executor(self) -> ThreadPoolExecutor:
176:         """Create thread pool executor lazily."""
177:         if self._executor is None:
178:             self._executor = ThreadPoolExecutor(
179:                 max_workers=self.config.max_workers,
180:                 thread_name_prefix="WorkerPool"
181:             )
182:         return self._executor
183:
184:     def _calculate_backoff_delay(self, retry_count: int) -> float:
185:         """
186:             Calculate exponential backoff delay.
187:
188:             Args:
189:                 retry_count: Number of retries already attempted
190:
191:             Returns:
192:                 Delay in seconds, capped at backoff_max_seconds
193:         """
194:         delay = self.config.backoff_base_seconds * (2 ** retry_count)
195:         return min(delay, self.config.backoff_max_seconds)
196:
197:     def _execute_task_with_retry(
198:         self,
199:         task_id: str,
200:         task_name: str,
201:         task_fn: Callable,
202:         args: tuple,
203:         kwargs: dict,
204:     ) -> Any:
205:         """
206:             Execute task with retry logic and exponential backoff.
207:
208:             Args:
209:                 task_id: Unique task identifier
210:                 task_name: Human-readable task name
211:                 task_fn: Task function to execute
212:                 args: Positional arguments for task_fn
213:                 kwargs: Keyword arguments for task_fn
214:
215:             Returns:
216:                 Task result
217:
218:             Raises:
219:                 TaskExecutionError: If task fails after all retries
220:         """
221:         worker_id = threading.current_thread().name
222:         retry_count = 0
223:         last_error = None
224:
```

```
225:     # Initialize metrics
226:     with self._lock:
227:         self._metrics[task_id] = TaskMetrics(
228:             task_id=task_id,
229:             task_name=task_name,
230:             status=TaskStatus.RUNNING,
231:             start_time=time.time(),
232:             worker_id=worker_id
233:         )
234:
235:     if self.config.enable_instrumentation:
236:         logger.info(f"[{task_id}] Starting task '{task_name}' on worker {worker_id}")
237:
238:     while retry_count <= self.config.max_retries:
239:         # Check if abort was requested
240:         if self._abort_requested.is_set():
241:             with self._lock:
242:                 self._metrics[task_id].status = TaskStatus.CANCELLED
243:                 self._metrics[task_id].end_time = time.time()
244:                 self._metrics[task_id].execution_time_ms = (
245:                     (self._metrics[task_id].end_time - self._metrics[task_id].start_time) * 1000
246:                 )
247:
248:             if self.config.enable_instrumentation:
249:                 logger.warning(f"[{task_id}] Task '{task_name}' cancelled due to abort request")
250:
251:             raise TaskExecutionError(f"Task {task_name} cancelled due to abort request")
252:
253:         try:
254:             # Execute task
255:             task_start = time.time()
256:             result = task_fn(*args, **kwargs)
257:             task_duration = (time.time() - task_start) * 1000
258:
259:             # Update metrics on success
260:             with self._lock:
261:                 self._metrics[task_id].status = TaskStatus.COMPLETED
262:                 self._metrics[task_id].end_time = time.time()
263:                 self._metrics[task_id].execution_time_ms = task_duration
264:                 self._metrics[task_id].retries_used = retry_count
265:
266:                 if self.config.enable_instrumentation:
267:                     logger.info(
268:                         f"[{task_id}] Task '{task_name}' completed successfully "
269:                         f"in {task_duration:.2f}ms (retries: {retry_count})"
270:                     )
271:
272:             return result
273:
274:         except Exception as e:
275:             last_error = e
276:
277:             # Update metrics on failure
278:             with self._lock:
279:                 self._metrics[task_id].retries_used = retry_count
280:                 self._metrics[task_id].error_message = str(e)
```

```
281:             if retry_count < self.config.max_retries:
282:                 # Calculate backoff delay
283:                 backoff_delay = self._calculate_backoff_delay(retry_count)
284:
285:
286:                 with self._lock:
287:                     self._metrics[task_id].status = TaskStatus.RETRYING
288:
289:                     if self.config.enable_instrumentation:
290:                         logger.warning(
291:                             f"[{task_id}] Task '{task_name}' failed (attempt {retry_count + 1}), "
292:                             f"retrying after {backoff_delay:.2f}s: {e}"
293:                         )
294:
295:                     # Wait before retrying (check abort periodically)
296:                     time.sleep(backoff_delay)
297:                     retry_count += 1
298:
299:                 else:
300:                     # All retries exhausted
301:                     with self._lock:
302:                         self._metrics[task_id].status = TaskStatus.FAILED
303:                         self._metrics[task_id].end_time = time.time()
304:                         self._metrics[task_id].execution_time_ms = (
305:                             (self._metrics[task_id].end_time - self._metrics[task_id].start_time) * 1000
306:                         )
307:
308:                     if self.config.enable_instrumentation:
309:                         logger.error(
310:                             f"[{task_id}] Task '{task_name}' failed after {retry_count} retries: {e}"
311:                         )
312:
313:                     raise TaskExecutionError(
314:                         f"Task {task_name} failed after {retry_count} retries: {last_error}"
315:                     ) from last_error
316:
317:             # Should not reach here, but just in case
318:             raise TaskExecutionError(f"Task {task_name} failed: {last_error}")
319:
320:     def submit_task(
321:         self,
322:         task_name: str,
323:         task_fn: Callable,
324:         args: tuple = (),
325:         kwargs: dict[str, Any] | None = None,
326:     ) -> str:
327:         """
328:             Submit a task for execution.
329:
330:             Args:
331:                 task_name: Human-readable task name for logging
332:                 task_fn: Callable to execute
333:                 args: Positional arguments for task_fn
334:                 kwargs: Keyword arguments for task_fn
335:
336:             Returns:
337:                 Unique task identifier
```

```
337:
338:     Raises:
339:         RuntimeError: If pool is shutdown
340:     """
341:     if self._is_shutdown:
342:         raise RuntimeError("Cannot submit tasks to a shutdown WorkerPool")
343:
344:     kwargs = kwargs or {}
345:     task_id = str(uuid4())
346:
347:     with self._lock:
348:         # Store task info for potential retries
349:         self._task_info[task_id] = (task_name, task_fn, args, kwargs)
350:
351:         # Submit task to executor
352:         executor = self._create_executor()
353:         future = executor.submit(
354:             self._execute_task_with_retry,
355:             task_id,
356:             task_name,
357:             task_fn,
358:             args,
359:             kwargs
360:         )
361:         self._futures[task_id] = future
362:
363:         if self.config.enable_instrumentation:
364:             logger.debug(f"[{task_id}] Task '{task_name}' submitted to pool")
365:
366:     return task_id
367:
368: def get_task_result(self, task_id: str, timeout: float | None = None) -> TaskResult:
369:     """
370:         Get result of a specific task.
371:
372:         Args:
373:             task_id: Task identifier returned by submit_task
374:             timeout: Maximum time to wait for result in seconds (None = wait forever)
375:
376:         Returns:
377:             TaskResult with execution metrics
378:
379:         Raises:
380:             KeyError: If task_id is not found
381:             TimeoutError: If timeout is exceeded
382:     """
383:     with self._lock:
384:         if task_id not in self._futures:
385:             raise KeyError(f"Task {task_id} not found")
386:
387:         future = self._futures[task_id]
388:         task_name = self._task_info[task_id][0]
389:
390:     try:
391:         timeout_to_use = timeout or self.config.task_timeout_seconds
392:         result = future.result(timeout=timeout_to_use)
```

```
393:
394:         with self._lock:
395:             metrics = self._metrics.get(task_id)
396:
397:             return TaskResult(
398:                 task_id=task_id,
399:                 task_name=task_name,
400:                 success=True,
401:                 result=result,
402:                 metrics=metrics
403:             )
404:
405:     except TimeoutError as e:
406:         with self._lock:
407:             metrics = self._metrics.get(task_id)
408:             if metrics:
409:                 metrics.status = TaskStatus.FAILED
410:                 metrics.error_message = f"Timeout after {timeout_to_use}s"
411:
412:             return TaskResult(
413:                 task_id=task_id,
414:                 task_name=task_name,
415:                 success=False,
416:                 error=e,
417:                 metrics=metrics
418:             )
419:
420:     except Exception as e:
421:         with self._lock:
422:             metrics = self._metrics.get(task_id)
423:
424:             return TaskResult(
425:                 task_id=task_id,
426:                 task_name=task_name,
427:                 success=False,
428:                 error=e,
429:                 metrics=metrics
430:             )
431:
432:     def wait_for_all(
433:         self,
434:         timeout: float | None = None,
435:         return_when: str = "ALL_COMPLETED"
436:     ) -> list[TaskResult]:
437:         """
438:             Wait for all submitted tasks to complete.
439:
440:             Args:
441:                 timeout: Maximum time to wait in seconds (None = wait forever)
442:                 return_when: When to return - "ALL_COMPLETED" or "FIRST_EXCEPTION"
443:
444:             Returns:
445:                 List of TaskResults for all tasks
446:
447:             Raises:
448:                 TimeoutError: If timeout is exceeded before all tasks complete
```

```
449:
450:     """
451:     if self.config.enable_instrumentation:
452:         logger.info(f"Waiting for {len(self._futures)} tasks to complete...")
453:
454:     start_time = time.time()
455:     results = []
456:
457:     with self._lock:
458:         all_futures = list(self._futures.items())
459:
460:     try:
461:         # Use as_completed for better progress tracking
462:         completed_count = 0
463:         for future in as_completed(
464:             [f for _, f in all_futures],
465:             timeout=timeout
466:         ):
467:             completed_count += 1
468:
469:             # Find task_id for this future
470:             task_id = None
471:             with self._lock:
472:                 for tid, f in all_futures:
473:                     if f == future:
474:                         task_id = tid
475:                         break
476:
477:             if task_id:
478:                 result = self.get_task_result(task_id, timeout=0.1)
479:                 results.append(result)
480:
481:                 if self.config.enable_instrumentation and completed_count % 10 == 0:
482:                     elapsed = time.time() - start_time
483:                     logger.info(
484:                         f"Progress: {completed_count}/{len(all_futures)} tasks completed"
485:                         f"\n{elapsed:.2f}s elapsed"
486:                     )
487:
488:             # Check if we should return early on first exception
489:             if return_when == "FIRST_EXCEPTION" and not result.success:
490:                 if self.config.enable_instrumentation:
491:                     logger.warning(
492:                         f"Returning early due to task failure: {result.task_name}"
493:                     )
494:                 break
495:
496:             elapsed = time.time() - start_time
497:             if self.config.enable_instrumentation:
498:                 successful = sum(1 for r in results if r.success)
499:                 failed = sum(1 for r in results if not r.success)
500:                 logger.info(
501:                     f"All tasks completed: {successful} succeeded, {failed} failed"
502:                     f"\n{elapsed:.2f}s total"
503:                 )
504:
505:     return results
```

```
505:
506:     except TimeoutError:
507:         elapsed = time.time() - start_time
508:         completed = len(results)
509:         pending = len(all_futures) - completed
510:
511:         logger.error(
512:             f"Timeout after {elapsed:.2f}s: {completed} completed, {pending} pending"
513:         )
514:
515:         # Get results for completed tasks
516:         for task_id, future in all_futures:
517:             if future.done() and task_id not in [r.task_id for r in results]:
518:                 try:
519:                     results.append(self.get_task_result(task_id, timeout=0.1))
520:                 except Exception as e:
521:                     logger.exception(f"Failed to get result for completed task {task_id}: {e}")
522:
523:         raise TimeoutError(
524:             f"Timeout waiting for tasks: {completed}/{len(all_futures)} completed"
525:         )
526:
527:     def abort_pending_tasks(self) -> int:
528:         """
529:             Request abort of all pending tasks.
530:
531:             This sets the abort flag, which will be checked by running tasks
532:             at their next safe point (before retry or next iteration).
533:
534:             Returns:
535:                 Number of tasks that were still pending
536:         """
537:         self._abort_requested.set()
538:
539:         pending_count = 0
540:         with self._lock:
541:             for task_id, future in self._futures.items():
542:                 if not future.done():
543:                     future.cancel()
544:                     pending_count += 1
545:
546:                     # Update metrics
547:                     if task_id in self._metrics:
548:                         self._metrics[task_id].status = TaskStatus.CANCELLED
549:
550:             if self.config.enable_instrumentation:
551:                 logger.warning(f"Abort requested: {pending_count} tasks cancelled")
552:
553:         return pending_count
554:
555:     def get_metrics(self) -> dict[str, TaskMetrics]:
556:         """
557:             Get execution metrics for all tasks.
558:
559:             Returns:
560:                 Dictionary mapping task_id to TaskMetrics

```

```
561:     """
562:     with self._lock:
563:         return dict(self._metrics)
564:
565:     def get_summary_metrics(self) -> dict[str, Any]:
566:         """
567:             Get summary metrics for the pool.
568:
569:             Returns:
570:                 Dictionary with aggregated metrics
571:             """
572:             with self._lock:
573:                 metrics_list = list(self._metrics.values())
574:
575:             if not metrics_list:
576:                 return {
577:                     "total_tasks": 0,
578:                     "completed": 0,
579:                     "failed": 0,
580:                     "cancelled": 0,
581:                     "running": 0,
582:                     "pending": 0,
583:                     "avg_execution_time_ms": 0.0,
584:                     "total_retries": 0,
585:                 }
586:
587:             completed = sum(1 for m in metrics_list if m.status == TaskStatus.COMPLETED)
588:             failed = sum(1 for m in metrics_list if m.status == TaskStatus.FAILED)
589:             cancelled = sum(1 for m in metrics_list if m.status == TaskStatus.CANCELLED)
590:             running = sum(1 for m in metrics_list if m.status == TaskStatus.RUNNING)
591:             pending = sum(1 for m in metrics_list if m.status == TaskStatus.PENDING)
592:
593:             completed_tasks = [m for m in metrics_list if m.status == TaskStatus.COMPLETED]
594:             avg_time = (
595:                 sum(m.execution_time_ms for m in completed_tasks) / len(completed_tasks)
596:                 if completed_tasks else 0.0
597:             )
598:
599:             total_retries = sum(m.retries_used for m in metrics_list)
600:
601:             return {
602:                 "total_tasks": len(metrics_list),
603:                 "completed": completed,
604:                 "failed": failed,
605:                 "cancelled": cancelled,
606:                 "running": running,
607:                 "pending": pending,
608:                 "avg_execution_time_ms": avg_time,
609:                 "total_retries": total_retries,
610:             }
611:
612:     def shutdown(self, wait: bool = True, cancel_futures: bool = False) -> None:
613:         """
614:             Shutdown the worker pool.
615:
616:             Args:
```

```
617:         wait: If True, wait for all tasks to complete before shutdown
618:         cancel_futures: If True, cancel all pending tasks
619:         """
620:     if self._is_shutdown:
621:         return
622:
623:     if cancel_futures:
624:         self.abort_pending_tasks()
625:
626:     if self._executor is not None:
627:         if self.config.enable_instrumentation:
628:             logger.info(f"Shutting down WorkerPool (wait={wait})")
629:
630:         self._executor.shutdown(wait=wait, cancel_futures=cancel_futures)
631:         self._executor = None
632:
633:     self._is_shutdown = True
634:
635:     if self.config.enable_instrumentation:
636:         summary = self.get_summary_metrics()
637:         logger.info(
638:             f"WorkerPool shutdown complete. "
639:             f"Completed: {summary['completed']}, "
640:             f"Failed: {summary['failed']}, "
641:             f"Cancelled: {summary['cancelled']} "
642:         )
643:
644:     def __enter__(self):
645:         """Context manager entry."""
646:         return self
647:
648:     def __exit__(self, exc_type, exc_val, exc_tb):
649:         """Context manager exit."""
650:         self.shutdown(wait=True)
651:         return False
652:
653:
654:
655: =====
656: FILE: src/farfan_pipeline/config/__init__.py
657: =====
658:
659:
660:
661:
662: =====
663: FILE: src/farfan_pipeline/config/dependency_lockdown.py
664: =====
665:
666: """
667: Dependency Lockdown
668:
669: Explicit allowlist of third-party modules permitted in F.A.R.F.A.N pipeline.
670:
671: This is the SINGLE SOURCE OF TRUTH for:
672: - Which third-party packages are allowed to be imported.
```

```
673: - Which dynamic imports (if any) are whitelisted.
674:
675: Maximum hardness interpretation:
676: - If a module is not in ALLOWED_THIRD_PARTY_MODULES, it MUST NOT be imported.
677: - If this file is missing or ALLOWED_THIRD_PARTY_MODULES is empty, policy_builder MUST fail hard.
678: - No silent fallback to requirements.txt is permitted.
679: """
680:
681: from typing import FrozenSet
682:
683: # Third-party modules explicitly allowed for import
684: # These MUST correspond to packages in requirements.txt but are explicitly vetted
685: ALLOWED_THIRD_PARTY_MODULES: FrozenSet[str] = frozenset(
686:     {
687:         # Core data science
688:         "numpy",
689:         "np",
690:         "pandas",
691:         "pd",
692:         "scipy",
693:         "sklearn",
694:         "scikit-learn",
695:         "scikit_learn",
696:         # Deep learning
697:         "torch",
698:         "pytorch",
699:         "tensorflow",
700:         "tf",
701:         "keras",
702:         "tf_keras",
703:         # NLP / Transformers
704:         "transformers",
705:         "sentence_transformers",
706:         "sentencepiece",
707:         "tokenizers",
708:         # Vector DB / RAG
709:         "chromadb",
710:         "chroma",
711:         "faiss",
712:         # LLM integrations
713:         "langchain",
714:         "langchain_core",
715:         "langchain_community",
716:         "openai",
717:         "anthropic",
718:         # Web frameworks
719:         "pydantic",
720:         "pydantic_core",
721:         "fastapi",
722:         "uvicorn",
723:         "starlette",
724:         # Testing
725:         "pytest",
726:         "hypothesis",
727:         "mock",
728:         # Utilities
```

```
729:     "click",
730:     "tqdm",
731:     "requests",
732:     "httpx",
733:     "aiohttp",
734:     "psutil",
735:     "pypdf",
736:     "pypdf2",
737:     "pdfplumber",
738:     "reportlab",
739:     # Data formats
740:     "yaml",
741:     "pyyaml",
742:     "toml",
743:     "tomli",
744:     "msgpack",
745:     # Serialization
746:     "orjson",
747:     "ujson",
748:     # Async
749:     "asyncio",
750:     "aiofiles",
751:     # Environment
752:     "dotenv",
753:     "python-dotenv",
754:     "python_dotenv",
755:     # Logging / Observability
756:     "loguru",
757:     "structlog",
758:     # Type checking (dev)
759:     "typing_extensions",
760:     "mypy",
761:     "mypy_extensions",
762:     # Linting (dev)
763:     "ruff",
764:     "black",
765:     "isort",
766:     # Documentation (dev)
767:     "sphinx",
768:     "mkdocs",
769:     # Packaging
770:     "setuptools",
771:     "wheel",
772:     "pip",
773:     "pipdeptree",
774:     # Deprecated warnings for migration
775:     "deprecated",
776:     "warnings",
777: }
778: )
779:
780: # Dynamic imports explicitly allowed (optional, default empty)
781: # These are module names that can be imported via importlib.import_module()
782: ALLOWED_DYNAMIC_IMPORTS: FrozenSet[str] = frozenset(
783:     {
784:         # Add dynamic imports here if needed
```

```
785:         # Example: "farfan_core.plugins.optional_module"
786:     }
787: )
788:
789:
790:
791: =====
792: FILE: src/farfan_pipeline/config/paths.py
793: =====
794:
795: """
796: Centralized Path Configuration for F.A.R.F.A.N
797: =====
798:
799: This module provides a single source of truth for all filesystem paths
800: used throughout the project. This ensures:
801:
802: 1. Portability: Works in development and production
803: 2. Configurability: Paths can be overridden via environment variables
804: 3. Consistency: All modules use the same path definitions
805: 4. Testability: Paths can be mocked for testing
806:
807: Author: Python Pipeline Expert
808: Date: 2025-11-15
809:
810: Usage:
811:     from farfan_core.config.paths import DATA_DIR, OUTPUT_DIR, CACHE_DIR
812:
813:     questionnaire = DATA_DIR / 'questionnaire_monolith.json'
814:     report = OUTPUT_DIR / 'analysis_report.json'
815: """
816:
817: import logging
818: import os
819: import sys
820: from pathlib import Path
821: from typing import Final, Tuple
822:
823: # =====
824: # Project Root Detection
825: # =====
826:
827: logger = logging.getLogger("farfan_core.config.paths")
828:
829:
830: def _detect_project_root() -> Tuple[Path, str]:
831:     """
832:         Detect project root directory reliably in both dev and production.
833:
834:     Strategy:
835:         1. If running from installed package: Use site-packages parent
836:         2. If running from source: Navigate from this file to project root
837:         3. Fallback: Use environment variable SAAAAAA_PROJECT_ROOT
838:     """
839:         # Check environment variable first (explicit override)
840:         if env_root := os.getenv('SAAAAAA_PROJECT_ROOT'):
```

```
841:         return Path(env_root).resolve(), "env"
842:
843:     # Detect from this file's location
844:     # src/farfan_core/config/paths.py \206\222 project_root
845:     this_file = Path(__file__).resolve()
846:
847:     # Navigate up: paths.py -> config -> farfan_core -> src -> project_root
848:     candidate = this_file.parents[3]
849:
850:     # Verify this looks like our project (has setup.py or pyproject.toml)
851:     if (candidate / 'setup.py').exists() or (candidate / 'pyproject.toml').exists():
852:         return candidate, "markers"
853:
854:     raise RuntimeError(
855:         "Unable to determine project root. "
856:         "Set the SAAAAAA_PROJECT_ROOT environment variable."
857:     )
858:
859:
860: # Project root (base for all other paths)
861: PROJECT_ROOT, PROJECT_ROOT_SOURCE = _detect_project_root()
862: logger.info("Project root detected via %s: %s", PROJECT_ROOT_SOURCE, PROJECT_ROOT)
863:
864: # =====
865: # Core Directories
866: # =====
867:
868: # Source code directory
869: SRC_DIR: Final[Path] = PROJECT_ROOT / 'src' / 'farfan_core'
870:
871: # Package root (for importlib.resources)
872: PACKAGE_ROOT: Final[Path] = SRC_DIR
873:
874: # =====
875: # Data Directories (Configurable)
876: # =====
877:
878: # Input data directory
879: DATA_DIR: Final[Path] = Path(
880:     os.getenv('SAAAAAA_DATA_DIR', str(PROJECT_ROOT / 'data'))
881: ).resolve()
882:
883: # Output directory for generated reports
884: OUTPUT_DIR: Final[Path] = Path(
885:     os.getenv('SAAAAAA_OUTPUT_DIR', str(PROJECT_ROOT / 'output'))
886: ).resolve()
887:
888: # Cache directory for temporary artifacts
889: CACHE_DIR: Final[Path] = Path(
890:     os.getenv('SAAAAAA_CACHE_DIR', str(PROJECT_ROOT / '.cache'))
891: ).resolve()
892:
893: # Logs directory
894: LOGS_DIR: Final[Path] = Path(
895:     os.getenv('SAAAAAA_LOGS_DIR', str(PROJECT_ROOT / 'logs'))
896: ).resolve()
```

```
897:  
898: # =====  
899: # Configuration Directories  
900: # =====  
901:  
902: # Configuration files directory  
903: CONFIG_DIR: Final[Path] = SRC_DIR / 'config'  
904:  
905: # Rules and schemas directory  
906: RULES_DIR: Final[Path] = PROJECT_ROOT / 'config' / 'rules'  
907: SCHEMAS_DIR: Final[Path] = PROJECT_ROOT / 'config' / 'schemas'  
908:  
909: # =====  
910: # Common Data Files  
911: # =====  
912:  
913: # Questionnaire monolith (canonical)  
914: QUESTIONNAIRE_FILE: Final[Path] = DATA_DIR / 'questionnaire_monolith.json'  
915:  
916: # Method catalog  
917: METHOD_CATALOG_FILE: Final[Path] = DATA_DIR / 'metodos' / 'catalogo_metodos.json'  
918:  
919: # =====  
920: # Test Directories  
921: # =====  
922:  
923: # Test data directory (fixtures, golden files, etc.)  
924: TEST_DATA_DIR: Final[Path] = PROJECT_ROOT / 'tests' / 'data'  
925:  
926: # Test output directory (temporary outputs from tests)  
927: TEST_OUTPUT_DIR: Final[Path] = PROJECT_ROOT / 'tests' / 'output'  
928:  
929: # =====  
930: # Utilities  
931: # =====  
932:  
933: def ensure_directories_exist() -> None:  
934:     """  
935:         Create all required directories if they don't exist.  
936:  
937:             This should be called at application startup to ensure the  
938:                 filesystem is properly initialized.  
939:     """  
940:     required_dirs = [  
941:         DATA_DIR,  
942:         OUTPUT_DIR,  
943:         CACHE_DIR,  
944:         LOGS_DIR,  
945:         TEST_OUTPUT_DIR,  
946:     ]  
947:  
948:     for dir_path in required_dirs:  
949:         dir_path.mkdir(parents=True, exist_ok=True)  
950:  
951:  
952: def get_output_path(plan_name: str, suffix: str = '') -> Path:
```

```
953: """
954:     Get output path for a specific plan analysis.
955:
956:     Args:
957:         plan_name: Name of the plan (e.g., "cpp_plan_1")
958:         suffix: Optional suffix for the output file
959:
960:     Returns:
961:         Path to output file
962:
963:     Example:
964:         >>> output_path = get_output_path("cpp_plan_1", "micro_analysis.json")
965:         >>> # Returns: output/cpp_plan_1/micro_analysis.json
966: """
967: plan_dir = OUTPUT_DIR / plan_name
968: plan_dir.mkdir(parents=True, exist_ok=True)
969:
970: if suffix:
971:     return plan_dir / suffix
972: return plan_dir
973:
974:
975: def get_cache_path(namespace: str, key: str) -> Path:
976: """
977:     Get cache path for a specific namespace and key.
978:
979:     Args:
980:         namespace: Cache namespace (e.g., "embeddings", "chunks")
981:         key: Cache key (will be sanitized)
982:
983:     Returns:
984:         Path to cache file
985:
986:     Example:
987:         >>> cache_path = get_cache_path("embeddings", "plan_123_chunk_5")
988:         >>> # Returns: .cache/embeddings/plan_123_chunk_5
989: """
990: namespace_dir = CACHE_DIR / namespace
991: namespace_dir.mkdir(parents=True, exist_ok=True)
992:
993: # Sanitize key (remove dangerous characters)
994: safe_key = key.replace('/', '_').replace('\\', '_').replace('..', '_')
995:
996: return namespace_dir / safe_key
997:
998:
999: def validate_paths() -> bool:
1000: """
1001:     Validate that all critical paths exist and are accessible.
1002:
1003:     Returns:
1004:         True if all paths are valid, False otherwise
1005: """
1006: issues = []
1007:
1008: # Check PROJECT_ROOT
```

```
1009:     if not PROJECT_ROOT.exists():
1010:         issues.append(f"PROJECT_ROOT does not exist: {PROJECT_ROOT}")
1011:
1012:     # Check SRC_DIR
1013:     if not SRC_DIR.exists():
1014:         issues.append(f"SRC_DIR does not exist: {SRC_DIR}")
1015:
1016:     # Check critical data files
1017:     if not QUESTIONNAIRE_FILE.exists():
1018:         issues.append(f"Questionnaire file not found: {QUESTIONNAIRE_FILE}")
1019:
1020:     if issues:
1021:         print("â\232 i,\217 Path validation issues:", file=sys.stderr)
1022:         for issue in issues:
1023:             print(f"    - {issue}", file=sys.stderr)
1024:         return False
1025:
1026:     return True
1027:
1028:
1029: # =====
1030: # Initialization
1031: # =====
1032:
1033: # Ensure directories exist on import (safe, idempotent)
1034: ensure_directories_exist()
1035:
1036: # =====
1037: # Compatibility Shims (DEPRECATED - for migration period)
1038: # =====
1039:
1040: # These provide backward compatibility during migration
1041: # TODO: Remove these after migration is complete
1042:
1043: def proj_root() -> Path:
1044:     """DEPRECATED: Use PROJECT_ROOT instead."""
1045:     import warnings
1046:     warnings.warn(
1047:         "proj_root() is deprecated. Use PROJECT_ROOT constant instead.",
1048:         DeprecationWarning,
1049:         stacklevel=2
1050:     )
1051:     return PROJECT_ROOT
1052:
1053:
1054: def reports_dir() -> Path:
1055:     """DEPRECATED: Use OUTPUT_DIR instead."""
1056:     import warnings
1057:     warnings.warn(
1058:         "reports_dir() is deprecated. Use OUTPUT_DIR constant instead.",
1059:         DeprecationWarning,
1060:         stacklevel=2
1061:     )
1062:     return OUTPUT_DIR
1063:
1064:
```

```
1065: # =====
1066: # Debug Information
1067: # =====
1068:
1069: if __name__ == "__main__":
1070:     """Print path configuration for debugging."""
1071:     print("=" * 80)
1072:     print("F.A.R.F.A.N Path Configuration")
1073:     print("=" * 80)
1074:     print()
1075:     print(f"PROJECT_ROOT: {PROJECT_ROOT}")
1076:     print(f"SRC_DIR: {SRC_DIR}")
1077:     print(f"DATA_DIR: {DATA_DIR}")
1078:     print(f"OUTPUT_DIR: {OUTPUT_DIR}")
1079:     print(f"CACHE_DIR: {CACHE_DIR}")
1080:     print(f"LOGS_DIR: {LOGS_DIR}")
1081:     print()
1082:     print(f"QUESTIONNAIRE: {QUESTIONNAIRE_FILE}")
1083:     print(f" Exists: {QUESTIONNAIRE_FILE.exists()}")
1084:     print()
1085:     print("Validation:", "â\234\205 PASS" if validate_paths() else "â\235\214 FAIL")
1086:     print()
1087:
1088:
1089:
1090: =====
1091: FILE: src/farfan_pipeline/contracts/__init__.py
1092: =====
1093:
1094: """
1095: Contracts Package
1096: """
1097: # Expose contracts for easier import
1098: from farfan_pipeline.contracts.routing_contract import RoutingContract
1099: from farfan_pipeline.contracts.snapshot_contract import SnapshotContract
1100: from farfan_pipeline.contracts.context_immutability import ContextImmutabilityContract
1101: from farfan_pipeline.contracts.permutation_invariance import PermutationInvarianceContract
1102: from farfan_pipeline.contracts.budget_monotonicity import BudgetMonotonicityContract
1103: from farfan_pipeline.contracts.total_ordering import TotalOrderingContract
1104: from farfan_pipeline.contracts.retriever_contract import RetrieverContract
1105: from farfan_pipeline.contracts.alignment_stability import AlignmentStabilityContract
1106: from farfan_pipeline.contracts.idempotency_dedup import IdempotencyContract
1107: from farfan_pipeline.contracts.risk_certificate import RiskCertificateContract
1108: from farfan_pipeline.contracts.monotone_compliance import MonotoneComplianceContract
1109: from farfan_pipeline.contracts.failureFallback import FailureFallbackContract
1110: from farfan_pipeline.contracts.concurrency_determinism import ConcurrencyDeterminismContract
1111: from farfan_pipeline.contracts.traceability import TraceabilityContract
1112: from farfan_pipeline.contracts.refusal import RefusalContract
1113:
1114:
1115:
1116: =====
1117: FILE: src/farfan_pipeline/contracts/alignment_stability.py
1118: =====
1119:
1120: """
```

```
1121: Alignment Stability Contract (ASC) - Implementation
1122: """
1123: import hashlib
1124: import json
1125: from typing import List, Dict, Any, Tuple
1126:
1127: class AlignmentStabilityContract:
1128:     @staticmethod
1129:         def compute_alignment(
1130:             sections: List[str],
1131:             standards: List[str],
1132:             params: Dict[str, Any]
1133:         ) -> Dict[str, Any]:
1134:             """
1135:                 Simulates Optimal Transport (EGW) alignment.
1136:                 In a real system, this would use POT (Python Optimal Transport).
1137:             """
1138:                 # Deterministic simulation based on inputs
1139:                 input_str = f"{sections}:{standards}:{json.dumps(params, sort_keys=True)}"
1140:                 hasher = hashlib.blake2b(input_str.encode(), digest_size=32)
1141:                 digest = hasher.hexdigest()
1142:
1143:                 # Simulate a plan (matrix) digest
1144:                 plan_digest = hashlib.blake2b(f"plan_{digest}".encode()).hexdigest()
1145:
1146:                 # Simulate cost and unmatched mass
1147:                 cost = int(digest[:4], 16) / 1000.0
1148:                 unmatched_mass = int(digest[4:8], 16) / 10000.0
1149:
1150:                 return {
1151:                     "plan_digest": plan_digest,
1152:                     "cost": cost,
1153:                     "unmatched_mass": unmatched_mass
1154:                 }
1155:
1156:     @staticmethod
1157:         def verify_stability(
1158:             sections: List[str],
1159:             standards: List[str],
1160:             params: Dict[str, Any]
1161:         ) -> bool:
1162:             """
1163:                 Verifies reproducibility with fixed hyperparameters.
1164:             """
1165:                 res1 = AlignmentStabilityContract.compute_alignment(sections, standards, params)
1166:                 res2 = AlignmentStabilityContract.compute_alignment(sections, standards, params)
1167:                 return res1 == res2
1168:
1169:
1170:
1171: =====
1172: FILE: src/farfan_pipeline/contracts/budget_monotonicity.py
1173: =====
1174:
1175: """
1176: Budget & Monotonicity Contract (BMC) - Implementation
```

```
1177: """
1178: from typing import List, Dict, Set
1179:
1180: class BudgetMonotonicityContract:
1181:     @staticmethod
1182:         def solve_knapsack(items: Dict[str, float], budget: float) -> Set[str]:
1183:             """
1184:                 Solves a knapsack-like problem (selecting items within budget).
1185:                 To ensure monotonicity ( $S^*(B_1) \supseteq S^*(B_2)$ ), we use a greedy approach based on cost/benefit
1186:                 or simply cost if benefit is uniform.
1187:                 Here we assume we want to maximize count of items, so we pick cheapest first.
1188:             """
1189:             sorted_items = sorted(items.items(), key=lambda x: x[1]) # Sort by cost ascending
1190:
1191:             selected = set()
1192:             current_cost = 0.0
1193:
1194:             for item_id, cost in sorted_items:
1195:                 if current_cost + cost <= budget:
1196:                     selected.add(item_id)
1197:                     current_cost += cost
1198:                 else:
1199:                     break
1200:
1201:             return selected
1202:
1203:     @staticmethod
1204:         def verify_monotonicity(items: Dict[str, float], budgets: List[float]) -> bool:
1205:             """
1206:                 Verifies  $S^*(B_1) \supseteq S^*(B_2)$  for  $B_1 < B_2$ .
1207:             """
1208:             sorted_budgets = sorted(budgets)
1209:             prev_solution = None
1210:
1211:             for b in sorted_budgets:
1212:                 solution = BudgetMonotonicityContract.solve_knapsack(items, b)
1213:                 if prev_solution is not None:
1214:                     if not prev_solution.issubset(solution):
1215:                         return False
1216:                     prev_solution = solution
1217:
1218:             return True
1219:
1220:
1221:
1222: =====
1223: FILE: src/farfan_pipeline/contracts/concurrency_determinism.py
1224: =====
1225:
1226: """
1227: Concurrency Determinism Contract (CDC) - Implementation
1228: """
1229: import hashlib
1230: import json
1231: import threading
1232: import time
```

```
1233: from typing import List, Any, Callable
1234:
1235: class ConcurrencyDeterminismContract:
1236:     @staticmethod
1237:         def execute_concurrently(
1238:             func: Callable[[Any], Any],
1239:             inputs: List[Any],
1240:             workers: int
1241:         ) -> List[Any]:
1242:             """
1243:                 Simulates concurrent execution.
1244:                 To ensure determinism, results must be sorted or indexed by input ID.
1245:             """
1246:             results = [None] * len(inputs)
1247:
1248:             def worker(idx, inp):
1249:                 # Simulate work
1250:                 time.sleep(0.001)
1251:                 results[idx] = func(inp)
1252:
1253:                 threads = []
1254:                 for i, inp in enumerate(inputs):
1255:                     t = threading.Thread(target=worker, args=(i, inp))
1256:                     threads.append(t)
1257:                     t.start()
1258:
1259:                     if len(threads) >= workers:
1260:                         for t in threads:
1261:                             t.join()
1262:                         threads = []
1263:
1264:                         for t in threads:
1265:                             t.join()
1266:
1267:             return results
1268:
1269:     @staticmethod
1270:         def verify_determinism(
1271:             func: Callable[[Any], Any],
1272:             inputs: List[Any]
1273:         ) -> bool:
1274:             """
1275:                 Verifies that 1 worker vs N workers produces hash-equal outputs.
1276:             """
1277:             res_seq = ConcurrencyDeterminismContract.execute_concurrently(func, inputs, workers=1)
1278:             res_conc = ConcurrencyDeterminismContract.execute_concurrently(func, inputs, workers=4)
1279:
1280:             hash1 = hashlib.blake2b(json.dumps(res_seq, sort_keys=True).encode()).hexdigest()
1281:             hash2 = hashlib.blake2b(json.dumps(res_conc, sort_keys=True).encode()).hexdigest()
1282:
1283:             return hash1 == hash2
1284:
1285:
1286:
1287: =====
1288: FILE: src/farfan_pipeline/contracts/context_immutability.py
```

```
1289: =====
1290:
1291: from __future__ import annotations
1292:
1293: import json
1294: from dataclasses import is_dataclass, fields, FrozenInstanceError
1295: from typing import Any
1296: from collections.abc import Mapping as ABCMapping
1297: from types import MappingProxyType
1298: from collections.abc import Sequence as ABCSequence # optional, for type checks
1299:
1300: class ContextImmutabilityContract:
1301:     @staticmethod
1302:     def _to_plain(obj: Any) -> Any:
1303:         """
1304:             Convert dataclasses + immutable containers (MappingProxyType, tuples, frozensets)
1305:             into plain Python structures without deepcopy(). This avoids mappingproxy pickling.
1306:         """
1307:         if is_dataclass(obj):
1308:             return {f.name: ContextImmutabilityContract._to_plain(getattr(obj, f.name)) for f in fields(obj)}
1309:         if isinstance(obj, (MappingProxyType, ABCMapping)):
1310:             return {k: ContextImmutabilityContract._to_plain(v) for k, v in obj.items()}
1311:         if isinstance(obj, (tuple, list, set, frozenset)):
1312:             return [ContextImmutabilityContract._to_plain(v) for v in obj]
1313:         return obj
1314:
1315:     @staticmethod
1316:     def _to_canonical(obj: Any) -> Any:
1317:         # Sort mapping keys deterministically; lists already deterministic after _to_plain
1318:         if isinstance(obj, dict):
1319:             return {k: ContextImmutabilityContract._to_canonical(obj[k]) for k in sorted(obj.keys())}
1320:         if isinstance(obj, list):
1321:             return [ContextImmutabilityContract._to_canonical(v) for v in obj]
1322:         return obj
1323:
1324:     @staticmethod
1325:     def canonical_digest(ctx: Any) -> str:
1326:         # Build plain JSON-safe object without deepcopy(), then canonicalize & hash.
1327:         plain = ContextImmutabilityContract._to_plain(ctx)
1328:         canon = ContextImmutabilityContract._to_canonical(plain)
1329:         s = json.dumps(canon, sort_keys=True, ensure_ascii=False, separators=(", ", ":" ))
1330:         try:
1331:             import blake3
1332:             return blake3.blake3(s.encode("utf-8")).hexdigest()
1333:         except Exception:
1334:             import hashlib
1335:             return hashlib.sha256(s.encode("utf-8")).hexdigest()
1336:
1337:     @staticmethod
1338:     def verify_immutability(ctx: Any) -> str:
1339:         """
1340:             Attempt to mutate: (1) top-level attribute, (2) deep mapping.
1341:             Both must fail. Return canonical digest for equality comparisons.
1342:         """
1343:         # 1) Top-level attribute mutation must fail
1344:         try:
```

```
1345:         setattr(ctx, "traceability_id", "MUTATE_ME")
1346:         raise RuntimeError("Mutation succeeded but should have failed!")
1347:     except (FrozenInstanceError, AttributeError, TypeError):
1348:         pass # expected
1349:
1350:     # 2) Deep mapping mutation must fail
1351:     deep_map = getattr(ctx, "dnp_standards", None)
1352:     if isinstance(deep_map, (MappingProxyType, ABCMapping)):
1353:         try:
1354:             deep_map["__MUTATE__"] = 1 # type: ignore[index]
1355:             raise RuntimeError("Deep mutation succeeded but should have failed!")
1356:         except (TypeError, AttributeError):
1357:             pass # expected
1358:
1359:     # 3) Deterministic canonical digest
1360:     return ContextImmutabilityContract.canonical_digest(ctx)
1361:
1362:
1363:
1364: =====
1365: FILE: src/farfan_pipeline/contracts/failureFallback.py
1366: =====
1367:
1368: """
1369: Failure & Fallback Contract (FFC) - Implementation
1370: """
1371: from typing import Callable, Any, Dict, Type, Tuple
1372:
1373: class FailureFallbackContract:
1374:     @staticmethod
1375:         def execute_with_fallback(
1376:             func: Callable,
1377:             fallback_value: Any,
1378:             expected_exceptions: Tuple[Type[Exception], ...]
1379:         ) -> Any:
1380:             """
1381:                 Executes func. If it raises an expected exception, returns fallback_value.
1382:                 Ensures determinism and no side effects (simulated).
1383:             """
1384:             try:
1385:                 return func()
1386:             except expected_exceptions:
1387:                 return fallback_value
1388:
1389:     @staticmethod
1390:         def verify_fallback_determinism(
1391:             func: Callable,
1392:             fallback_value: Any,
1393:             exception_type: Type[Exception]
1394:         ) -> bool:
1395:             """
1396:                 Verifies that repeated failures produce identical fallback values.
1397:             """
1398:             res1 = FailureFallbackContract.execute_with_fallback(func, fallback_value, (exception_type,))
1399:             res2 = FailureFallbackContract.execute_with_fallback(func, fallback_value, (exception_type,))
1400:             return res1 == res2
```

```
1401:  
1402:  
1403:  
1404: =====  
1405: FILE: src/farfan_pipeline/contracts/governance.py  
1406: =====  
1407:  
1408: """  
1409: Contract Governance Utilities  
1410: """  
1411: from typing import Type, Any, Callable  
1412: import functools  
1413:  
1414: def uses_contract(contract_class: Type[Any]) -> Callable:  
1415:     """  
1416:         Decorator to explicitly declare that a function or class relies on a specific contract.  
1417:         This serves as documentation and allows for static analysis of contract dependencies.  
1418:  
1419:         Usage:  
1420:             @uses_contract(RoutingContract)  
1421:             def my_function():  
1422:                 ...  
1423:             """  
1424:             def decorator(obj: Any) -> Any:  
1425:                 if not hasattr(obj, "_contract_dependencies"):  
1426:                     obj._contract_dependencies = []  
1427:                 obj._contract_dependencies.append(contract_class)  
1428:  
1429:                 # If it's a function, wrap it to preserve metadata  
1430:                 if callable(obj) and not isinstance(obj, type):  
1431:                     @functools.wraps(obj)  
1432:                     def wrapper(*args, **kwargs):  
1433:                         return obj(*args, **kwargs)  
1434:                     wrapper._contract_dependencies = obj._contract_dependencies  
1435:                     return wrapper  
1436:  
1437:                 return obj  
1438:             return decorator  
1439:  
1440:  
1441:  
1442: =====  
1443: FILE: src/farfan_pipeline/contracts/idempotency_dedup.py  
1444: =====  
1445:  
1446: """  
1447: Idempotency & De-dup Contract (IDC) - Implementation  
1448: """  
1449: import hashlib  
1450: import json  
1451: from typing import List, Dict, Any, Set  
1452:  
1453: class EvidenceStore:  
1454:     def __init__(self):  
1455:         self.evidence: Dict[str, Any] = {} # content_hash -> evidence  
1456:         self.duplicates_blocked = 0
```

```
1457:
1458:     def add(self, item: Dict[str, Any]):
1459:         # Calculate content hash
1460:         content_hash = hashlib.blake2b(json.dumps(item, sort_keys=True).encode()).hexdigest()
1461:
1462:         if content_hash in self.evidence:
1463:             self.duplicates_blocked += 1
1464:         else:
1465:             self.evidence[content_hash] = item
1466:
1467:     def state_hash(self) -> str:
1468:         # Hash of sorted keys to ensure order independence
1469:         sorted_keys = sorted(self.evidence.keys())
1470:         return hashlib.blake2b(json.dumps(sorted_keys).encode()).hexdigest()
1471:
1472: class IdempotencyContract:
1473:     @staticmethod
1474:     def verify_idempotency(items: List[Dict[str, Any]]) -> Dict[str, Any]:
1475:         store = EvidenceStore()
1476:         for item in items:
1477:             store.add(item)
1478:
1479:         return {
1480:             "state_hash": store.state_hash(),
1481:             "duplicates_blocked": store.duplicates_blocked,
1482:             "count": len(store.evidence)
1483:         }
1484:
1485:
1486:
1487: =====
1488: FILE: src/farfan_pipeline/contracts/monotone_compliance.py
1489: =====
1490:
1491: """
1492: Monotone Compliance Contract (MCC) - Implementation
1493: """
1494: from typing import Set, Dict, Any
1495: from enum import IntEnum
1496:
1497: class Label(IntEnum):
1498:     UNSAT = 0
1499:     PARTIAL = 1
1500:     SAT = 2
1501:
1502: class MonotoneComplianceContract:
1503:     @staticmethod
1504:     def evaluate(evidence: Set[str], rules: Dict[str, Any]) -> Label:
1505:         """
1506:             Evaluates label based on evidence and Horn-like clauses.
1507:             Simple logic:
1508:             - SAT if all 'sat_reqs' present
1509:             - PARTIAL if all 'partial_reqs' present
1510:             - UNSAT otherwise
1511:         """
1512:         sat_reqs = set(rules.get("sat_reqs", []))
```

```

1513:         partial_reqs = set(rules.get("partial_reqs", []))
1514:
1515:     if sat_reqs.issubset(evidence):
1516:         return Label.SAT
1517:     elif partial_reqs.issubset(evidence):
1518:         return Label.PARTIAL
1519:     else:
1520:         return Label.UNSAT
1521:
1522:     @staticmethod
1523:     def verify_monotonicity(
1524:         evidence_subset: Set[str],
1525:         evidence_superset: Set[str],
1526:         rules: Dict[str, Any]
1527:     ) -> bool:
1528:         """
1529:             Verifies label(E') >= label(E) for E ⊂ E'.
1530:         """
1531:         if not evidence_subset.issubset(evidence_superset):
1532:             raise ValueError("Subset is not contained in superset")
1533:
1534:         l1 = MonotoneComplianceContract.evaluate(evidence_subset, rules)
1535:         l2 = MonotoneComplianceContract.evaluate(evidence_superset, rules)
1536:
1537:         return l2 >= l1
1538:
1539:
1540:
1541: =====
1542: FILE: src/farfan_pipeline/contracts/permuation_invariance.py
1543: =====
1544:
1545: """
1546: Permutation-Invariance Contract (PIC) - Implementation
1547: """
1548: import hashlib
1549: from typing import List, Any, Callable
1550:
1551: class PermutationInvarianceContract:
1552:     @staticmethod
1553:     def aggregate(items: List[Any], transform: Callable[[Any], float]) -> float:
1554:         """
1555:             Implements f(S) = ∑_{x ∈ S} f(x) pattern for permutation invariance.
1556:             Here, sum is the aggregation function (symmetric).
1557:         """
1558:         # ∑_{x ∈ S} = transform(x)
1559:         transformed = [transform(x) for x in items]
1560:
1561:         # If ∑_{x ∈ S} = Sum is order-independent (within floating point limits, usually)
1562:         # For strict bitwise invariance with floats, we might need to sort or use exact arithmetic.
1563:         # But the requirement asks for "numerical tolerance".
1564:         total = sum(transformed)
1565:
1566:         # ∑_{x ∈ S} = identity (for this example)
1567:         return total
1568:
```

```
1569:     @staticmethod
1570:     def verify_invariance(items: List[Any], transform: Callable[[Any], float]) -> str:
1571:         """
1572:             Calculates digest of the aggregation.
1573:         """
1574:         result = PermutationInvarianceContract.aggregate(items, transform)
1575:         return hashlib.blake2b(str(result).encode()).hexdigest()
1576:
1577:
1578:
1579: =====
1580: FILE: src/farfan_pipeline/contracts/refusal.py
1581: =====
1582:
1583: """
1584: Refusal Contract (RefC) - Implementation
1585: """
1586: from typing import Dict, Any
1587:
1588: class RefusalError(Exception):
1589:     pass
1590:
1591: class RefusalContract:
1592:     @staticmethod
1593:     def check_prerequisites(context: Dict[str, Any]):
1594:         """
1595:             Confirma que ante prerequisitos fallidos el sistema rehÃ³sa con motivo tipado.
1596:         """
1597:         if "mandatory" not in context:
1598:             raise RefusalError("Missing mandatory field")
1599:
1600:         if context.get("alpha", 1.0) > 0.5:
1601:             raise RefusalError("Alpha violation")
1602:
1603:         if "sigma" not in context:
1604:             raise RefusalError("Sigma absent")
1605:
1606:     @staticmethod
1607:     def verify_refusal(context: Dict[str, Any]) -> str:
1608:         try:
1609:             RefusalContract.check_prerequisites(context)
1610:             return "OK"
1611:         except RefusalError as e:
1612:             return str(e)
1613:
1614:
1615:
1616: =====
1617: FILE: src/farfan_pipeline/contracts/retriever_contract.py
1618: =====
1619:
1620: """
1621: Retriever Contract (ReC) - Implementation
1622: """
1623: import hashlib
1624: import json
```

```
1625: from typing import List, Dict, Any
1626:
1627: class RetrieverContract:
1628:     @staticmethod
1629:         def retrieve(query: str, filters: Dict[str, Any], index_hash: str, top_k: int = 5) -> List[Dict[str, Any]]:
1630:             """
1631:                 Simulates hybrid retrieval (patterns+dimension+203+230).
1632:                 In a real system, this would call FAISS/Pyserini.
1633:                 Here we simulate deterministic retrieval based on inputs.
1634:             """
1635:                 # Deterministic simulation
1636:                 input_data = f"{query}:{json.dumps(filters, sort_keys=True)}:{index_hash}"
1637:                 hasher = hashlib.blake2b(input_data.encode(), digest_size=32)
1638:
1639:                 results = []
1640:                 current_hash = hasher.hexdigest()
1641:
1642:                 for i in range(top_k):
1643:                     doc_hash = hashlib.blake2b(f"{current_hash}:{i}".encode()).hexdigest()
1644:                     results.append({
1645:                         "id": f"doc_{doc_hash[:8]}",
1646:                         "score": 0.9 - (i * 0.1),
1647:                         "content_hash": doc_hash
1648:                     })
1649:
1650:                 return results
1651:
1652:     @staticmethod
1653:         def verify_determinism(query: str, filters: Dict[str, Any], index_hash: str) -> str:
1654:             """
1655:                 Returns a digest of the top-K results to verify determinism.
1656:             """
1657:                 results = RetrieverContract.retrieve(query, filters, index_hash)
1658:                 # De-dup by content_hash is implicit if retrieval is deterministic,
1659:                 # but we can enforce it here if needed.
1660:
1661:                 # Serialize results for hashing
1662:                 return hashlib.blake2b(json.dumps(results, sort_keys=True).encode()).hexdigest()
1663:
1664:
1665: =====
1666: =====
1667: FILE: src/farfan_pipeline/contracts/risk_certificate.py
1668: =====
1669:
1670: """
1671: Risk Certificate Contract (RCC) - Implementation
1672: """
1673: import numpy as np
1674: from typing import List, Tuple, Dict
1675:
1676: class RiskCertificateContract:
1677:     @staticmethod
1678:         def conformal_prediction(
1679:             calibration_scores: List[float],
1680:             alpha: float
```

```
1681:     ) -> float:
1682:         """
1683:             Computes the quantile for conformal prediction.
1684:             q = (1 - alpha) * (n + 1) / n corrected quantile
1685:         """
1686:         n = len(calibration_scores)
1687:         q_level = np.ceil((n + 1) * (1 - alpha)) / n
1688:         q_level = min(1.0, max(0.0, q_level))
1689:
1690:         # Use numpy quantile
1691:         return np.quantile(calibration_scores, q_level, method='higher')
1692:
1693:     @staticmethod
1694:     def verify_risk(
1695:         calibration_data: List[float],
1696:         holdout_data: List[float],
1697:         alpha: float,
1698:         seed: int
1699:     ) -> Dict[str, float]:
1700:         """
1701:             Verifies that empirical coverage is approx (1-alpha) and risk <= alpha.
1702:         """
1703:         np.random.seed(seed)
1704:
1705:         # Compute threshold from calibration data
1706:         threshold = RiskCertificateContract.conformal_prediction(calibration_data, alpha)
1707:
1708:         # Check coverage on holdout
1709:         covered = [s <= threshold for s in holdout_data]
1710:         coverage = sum(covered) / len(holdout_data)
1711:         risk = 1.0 - coverage
1712:
1713:         return {
1714:             "alpha": alpha,
1715:             "threshold": float(threshold),
1716:             "coverage": coverage,
1717:             "risk": risk
1718:         }
1719:
1720:
```