```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 19
================================================================================
Generated: 2025-12-07T06:17:24.378662
Files in this batch: 17
================================================================================


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

"""
FARFAN Mechanistic Policy Pipeline - Observability Module
==========================================================

OpenTelemetry-based observability for distributed tracing and monitoring.

Author: FARFAN Team
Date: 2025-11-13
Version: 1.0.0
"""

from farfan_pipeline.observability.opentelemetry_integration import (
    ExecutorSpanDecorator,
    OpenTelemetryObservability,
    Span,
    SpanContext,
    SpanKind,
    SpanStatus,
    Tracer,
    executor_span,
    get_global_observability,
    get_tracer,
)

__all__ = [
    "ExecutorSpanDecorator",
    "OpenTelemetryObservability",
    "Span",
    "SpanContext",
    "SpanKind",
    "SpanStatus",
    "Tracer",
    "executor_span",
    "get_global_observability",
    "get_tracer",
]


================================================================================
FILE: src/farfan_pipeline/observability/canonical_metrics/__init__.py
================================================================================

"""Canonical Metrics and Monitoring System.
```

```
57:
58: This module provides comprehensive health checks, metrics export, and
59: observability tools for the SAAAAAA orchestration system.
60: """
61:
62: from farfan_pipeline.observability.canonical_metrics.health import get_system_health
63: from farfan_pipeline.observability.canonical_metrics.metrics import export_metrics
64:
65: __all__ = ["get_system_health", "export_metrics"]
66:
67:
68:
69: ================================================================================
70: FILE: src/farfan_pipeline/observability/canonical_metrics/health.py
71: ================================================================================
72:
73: """System Health Check Module.
74:
75: Provides comprehensive health status for all system components.
76: """
77:
78: import logging
79: from datetime import datetime
80: from typing import Any
81:
82: logger = logging.getLogger(__name__)
83:
84:
85: def get_system_health(orchestrator: Any) -> dict[str, Any]:
86:     """
87:     Comprehensive system health check.
88:
89:     Args:
90:         orchestrator: The Orchestrator instance to check health for
91:
92:     Returns:
93:         Health status with component checks
94:     """
95:     health = {
96:         'status': 'healthy',
97:         'timestamp': datetime.utcnow().isoformat(),
98:         'components': {}
99:     }
100:
101:     # Check method executor
102:     try:
103:         if hasattr(orchestrator, 'executor'):
104:             executor_health = {
105:                 'instances_loaded': len(orchestrator.executor.instances),
106:                 'calibrations_loaded': len(orchestrator.executor.calibrations),
107:                 'status': 'healthy'
108:             }
109:             health['components']['method_executor'] = executor_health
110:         else:
111:             health['components']['method_executor'] = {
112:                 'status': 'unavailable',
```

```
                'error': 'No executor attribute found'
            }
    except Exception as e:
        health['status'] = 'unhealthy'
        health['components']['method_executor'] = {
            'status': 'unhealthy',
            'error': str(e)
        }

    # Check questionnaire provider
    try:
        from farfan_pipeline.core.orchestrator.questionnaire import get_questionnaire_provider
        provider = get_questionnaire_provider()
        questionnaire_health = {
            'has_data': provider.has_data(),
            'status': 'healthy' if provider.has_data() else 'unhealthy'
        }
        health['components']['questionnaire_provider'] = questionnaire_health

        if not provider.has_data():
            health['status'] = 'degraded'
    except Exception as e:
        health['status'] = 'unhealthy'
        health['components']['questionnaire_provider'] = {
            'status': 'unhealthy',
            'error': str(e)
        }

    # Check resource limits
    try:
        if hasattr(orchestrator, 'resource_limits'):
            usage = orchestrator.resource_limits.get_resource_usage()
            resource_health = {
                'cpu_percent': usage.get('cpu_percent', 0),
                'memory_mb': usage.get('rss_mb', 0),
                'worker_budget': usage.get('worker_budget', 0),
                'status': 'healthy'
            }

            # Warning thresholds
            if usage.get('cpu_percent', 0) > 80:
                resource_health['status'] = 'degraded'
                health['status'] = 'degraded'

            if usage.get('rss_mb', 0) > 3500:  # Near 4GB limit
                resource_health['status'] = 'degraded'
                health['status'] = 'degraded'

            health['components']['resources'] = resource_health
        else:
            health['components']['resources'] = {
                'status': 'unavailable',
                'error': 'No resource_limits attribute found'
            }
    except Exception as e:
        health['status'] = 'unhealthy'
```

```
        health['components']['resources'] = {
            'status': 'unhealthy',
            'error': str(e)
        }

    return health


================================================================================
FILE: src/farfan_pipeline/observability/canonical_metrics/metrics.py
================================================================================

"""Metrics Export Module.

Provides comprehensive metrics export for monitoring and observability.
"""

import logging
from datetime import datetime
from typing import Any

logger = logging.getLogger(__name__)


def export_metrics(orchestrator: Any) -> dict[str, Any]:
    """Export all metrics for monitoring.

    Args:
        orchestrator: The Orchestrator instance to export metrics from

    Returns:
        Dictionary containing all system metrics
    """
    metrics = {
        'timestamp': datetime.utcnow().isoformat(),
        'phase_metrics': {},
        'resource_usage': {},
        'abort_status': {},
        'phase_status': {},
    }

    # Export phase metrics
    try:
        if hasattr(orchestrator, 'get_phase_metrics'):
            metrics['phase_metrics'] = orchestrator.get_phase_metrics()
    except Exception as e:
        logger.error(f"Failed to export phase metrics: {e}")
        metrics['phase_metrics'] = {'error': str(e)}

    # Export resource usage history
    try:
        if hasattr(orchestrator, 'resource_limits') and hasattr(
            orchestrator.resource_limits, 'get_usage_history'
        ):
            metrics['resource_usage'] = orchestrator.resource_limits.get_usage_history()
```

```
    except Exception as e:
        logger.error(f"Failed to export resource usage: {e}")
        metrics['resource_usage'] = {'error': str(e)}

    # Export abort status
    try:
        if hasattr(orchestrator, 'abort_signal'):
            abort_signal = orchestrator.abort_signal
            metrics['abort_status'] = {
                'is_aborted': abort_signal.is_aborted(),
                'reason': abort_signal.get_reason() if hasattr(abort_signal, 'get_reason') else None,
                'timestamp': (
                    abort_signal.get_timestamp().isoformat()
                    if hasattr(abort_signal, 'get_timestamp') and abort_signal.get_timestamp()
                    else None
                ),
            }
    except Exception as e:
        logger.error(f"Failed to export abort status: {e}")
        metrics['abort_status'] = {'error': str(e)}

    # Export phase status
    try:
        if hasattr(orchestrator, '_phase_status'):
            metrics['phase_status'] = dict(orchestrator._phase_status)
    except Exception as e:
        logger.error(f"Failed to export phase status: {e}")
        metrics['phase_status'] = {'error': str(e)}

    return metrics



================================================================================
FILE: src/farfan_pipeline/observability/import_scanner.py
================================================================================

"""
Import Scanner

AST-based static analysis of import statements.
Validates imports against ImportPolicy without executing code.

"Maximum hardness" semantics:
- Any import that cannot be resolved to:
    - an internal farfan_core.* module, OR
    - a stdlib module, OR
    - an explicitly allowed third-party module
  is a violation.
- Any file that cannot be read or parsed is a violation.
- Any relative import that cannot be resolved to an absolute module name is a violation.
"""

from __future__ import annotations

import ast
```

```
281: from pathlib import Path
282: from typing import Iterable, Optional
283:
284: from farfan_pipeline.observability.path_import_policy import ImportPolicy, PolicyReport, PolicyViolation
285:
286:
287: def _is_stdlib_module(module_name: str, stdlib_modules: frozenset[str]) -> bool:
288:     """Check if module is from stdlib (module or any parent package)."""
289:     if module_name in stdlib_modules:
290:         return True
291:
292:     parts = module_name.split(".")
293:     for i in range(1, len(parts)):
294:         candidate = ".".join(parts[: i + 1])
295:         if candidate in stdlib_modules:
296:             return True
297:
298:     return False
299:
300:
301: def _is_internal_module(module_name: str, allowed_prefixes: frozenset[str]) -> bool:
302:     """Check if module is internal (starts with any allowed prefix)."""
303:     for prefix in allowed_prefixes:
304:         if module_name == prefix or module_name.startswith(prefix + "."):
305:             return True
306:     return False
307:
308:
309: def _is_allowed_third_party(module_name: str, allowed_third_party: frozenset[str]) -> bool:
310:     """Check if module is an allowed third-party module or submodule."""
311:     parts = module_name.split(".")
312:
313:     # Try longest prefix â\206\222 shortest (e.g. foo.bar.baz â\206\222 foo.bar.baz, foo.bar, foo)
314:     for i in range(len(parts), 0, -1):
315:         candidate = ".".join(parts[:i])
316:         if candidate in allowed_third_party:
317:             return True
318:
319:     # As a last resort, allow if any single segment is explicitly whitelisted
320:     for part in parts:
321:         if part in allowed_third_party:
322:             return True
323:
324:     return False
325:
326:
327: def _validate_import(
328:     module_name: str,
329:     policy: ImportPolicy,
330:     file_path: Path,
331:     line_number: int,
332: ) -> Optional[PolicyViolation]:
333:     """
334:     Validate a single import against policy.
335:
336:     Returns:
```

```
337:         PolicyViolation if import violates policy, None otherwise.
338:     """
339:     if not module_name:
340:         return None
341:
342:     # Internal?
343:     if _is_internal_module(module_name, policy.allowed_internal_prefixes):
344:         return None
345:
346:     # Stdlib?
347:     if _is_stdlib_module(module_name, policy.stdlib_modules):
348:         return None
349:
350:     # Third-party?
351:     if _is_allowed_third_party(module_name, policy.allowed_third_party):
352:         return None
353:
354:     # Otherwise: violation
355:     return PolicyViolation(
356:         kind="static_import",
357:         message=f"Import '{module_name}' not allowed by ImportPolicy",
358:         file=file_path,
359:         line=line_number,
360:         operation="import",
361:         target=module_name,
362:     )
363:
364:
365: def _module_name_from_path(py_file: Path, repo_root: Path) -> Optional[str]:
366:     """
367:     Compute fully-qualified module name for a given .py file
368:     assuming it lives under farfan_core/farfan_core.
369:
370:     Example:
371:         <REPO>/farfan_core/farfan_core/core/foo/bar.py
372:         -> farfan_core.core.foo.bar
373:     """
374:     try:
375:         rel = py_file.resolve().relative_to(repo_root)
376:     except Exception:
377:         return None
378:
379:     parts = rel.with_suffix("").parts  # drop ".py"
380:     if len(parts) >= 2 and parts[0] == "farfan_core" and parts[1] == "farfan_core":
381:         # ("farfan_core", "farfan_core", "core", "foo") -> "farfan_core.core.foo"
382:         return ".".join(("farfan_core",) + parts[2:])
383:     return None
384:
385:
386: def _resolve_from_import_module(
387:     node: ast.ImportFrom,
388:     current_module: Optional[str],
389: ) -> Optional[str]:
390:     """
391:     Resolve absolute module name for a 'from ... import ...' statement.
392:
```

```
    For absolute imports (level == 0), returns node.module.
    For relative imports (level > 0), uses current_module as base.

    If resolution is ambiguous or impossible, returns None.
    """
    if node.level == 0:
        # from x import y
        return node.module or None

    if current_module is None:
        # Cannot resolve relative import without knowing current module
        return None

    base_parts = current_module.split(".")
    if len(base_parts) < node.level:
        # from ... import beyond root: ambiguous
        return None

    parent_parts = base_parts[: -node.level]

    if node.module:
        parent_parts.extend(node.module.split("."))

    return ".".join(parent_parts)


def validate_imports(
    roots: Iterable[Path],
    import_policy: ImportPolicy,
    repo_root: Path,
) -> PolicyReport:
    """
    Scan Python files under roots and validate imports.

    Args:
        roots: Directories to scan for .py files.
        import_policy: Policy defining allowed imports.
        repo_root: Repository root path, used to resolve module names from paths.

    Returns:
        PolicyReport with any static_import_violations found.
    """
    report = PolicyReport()

    for root in roots:
        if not root.exists():
            continue

        for py_file in root.rglob("*.py"):
            # Skip __pycache__
            if "__pycache__" in py_file.parts:
                continue

            try:
                source = py_file.read_text(encoding="utf-8")
            except Exception as e:
```

```
                # Treat unreadable files as violations, not silent skips
                report.static_import_violations.append(
                    PolicyViolation(
                        kind="static_import",
                        message=f"Failed to read file for import scan: {e}",
                        file=py_file,
                        line=None,
                        operation="read_source",
                        target=str(py_file),
                    )
                )
                continue

            try:
                tree = ast.parse(source, filename=str(py_file))
            except SyntaxError as e:
                report.static_import_violations.append(
                    PolicyViolation(
                        kind="static_import",
                        message=f"Syntax error while parsing for import scan: {e.msg}",
                        file=py_file,
                        line=e.lineno or None,
                        operation="parse",
                        target=str(py_file),
                    )
                )
                continue
            except Exception as e:
                report.static_import_violations.append(
                    PolicyViolation(
                        kind="static_import",
                        message=f"Unexpected error while parsing for import scan: {e}",
                        file=py_file,
                        line=None,
                        operation="parse",
                        target=str(py_file),
                    )
                )
                continue

            current_module = _module_name_from_path(py_file, repo_root)

            for node in ast.walk(tree):
                if isinstance(node, ast.Import):
                    for alias in node.names:
                        module_name = alias.name
                        violation = _validate_import(
                            module_name,
                            import_policy,
                            py_file,
                            node.lineno,
                        )
                        if violation:
                            report.static_import_violations.append(violation)

                elif isinstance(node, ast.ImportFrom):
```

```
                    abs_module = _resolve_from_import_module(node, current_module)
                    if not abs_module:
                        # Could not resolve; treat as a hard violation
                        report.static_import_violations.append(
                            PolicyViolation(
                                kind="static_import",
                                message=(
                                    "Unable to resolve relative import "
                                    f"(module={node.module!r}, level={node.level})"
                                ),
                                file=py_file,
                                line=node.lineno,
                                operation="import_from",
                                target=node.module or "",
                            )
                        )
                        continue

                    violation = _validate_import(
                        abs_module,
                        import_policy,
                        py_file,
                        node.lineno,
                    )
                    if violation:
                        report.static_import_violations.append(violation)

    return report


================================================================================
FILE: src/farfan_pipeline/observability/no_relative_imports_contract.py
================================================================================

"""Import Linter contract that forbids relative imports in farfan_pipeline."""

from __future__ import annotations

import ast
import importlib
from pathlib import Path
from typing import Iterable, List, Sequence, Tuple

from grimp import ImportGraph

from importlinter.application import output
from importlinter.domain.contract import Contract, ContractCheck, InvalidContractOptions

RelativeImport = Tuple[str, int, str]


class NoRelativeImportsContract(Contract):
    """Contract that fails if any relative imports are present."""

    def check(self, graph: ImportGraph, verbose: bool) -> ContractCheck:
        package_paths = self._get_package_paths()
```

```
561:
562:         output.verbose_print(verbose, f"Scanning {len(package_paths)} root packages for relatives.")
563:
564:         violations: list[RelativeImport] = []
565:         for package_name, package_path in package_paths:
566:             output.verbose_print(verbose, f"  -> {package_name} ({package_path})")
567:             violations.extend(self._scan_package(package_path))
568:
569:         violations.sort(key=lambda item: (item[0], item[1]))
570:         return ContractCheck(kept=not violations, metadata={"violations": violations})
571:
572:     def render_broken_contract(self, check: ContractCheck) -> None:
573:         violations: Sequence[RelativeImport] = check.metadata.get("violations", [])
574:         output.print_error("Relative imports detected (use absolute farfan_pipeline.* paths):")
575:         for path, line, statement in violations:
576:             output.print(f"  {path}:{line}: {statement}")
577:
578:     def _get_package_paths(self) -> List[tuple[str, Path]]:
579:         """Resolve filesystem paths for configured root packages."""
580:         package_paths: list[tuple[str, Path]] = []
581:         for package_name in self.session_options.get("root_packages", []):
582:             try:
583:                 module = importlib.import_module(package_name)
584:             except ModuleNotFoundError as exc:
585:                 raise InvalidContractOptions(
586:                     {package_name: f"Cannot import root package '{package_name}': {exc}"}
587:                 )
588:
589:             module_file = getattr(module, "__file__", None)
590:             if not module_file:
591:                 raise InvalidContractOptions({package_name: "Root package has no __file__."})
592:
593:             package_paths.append((package_name, Path(module_file).resolve().parent))
594:         return package_paths
595:
596:     def _scan_package(self, package_path: Path) -> List[RelativeImport]:
597:         repo_root = self._find_repo_root(package_path)
598:         violations: list[RelativeImport] = []
599:
600:         for py_file in package_path.rglob("*.py"):
601:             if "__pycache__" in py_file.parts:
602:                 continue
603:
604:             for line_number, statement in self._find_relative_imports(py_file):
605:                 try:
606:                     display_path = str(py_file.relative_to(repo_root))
607:                 except ValueError:
608:                     display_path = str(py_file)
609:                 violations.append((display_path, line_number, statement))
610:
611:         return violations
612:
613:     def _find_relative_imports(self, py_file: Path) -> Iterable[tuple[int, str]]:
614:         try:
615:             content = py_file.read_text(encoding="utf-8")
616:         except Exception as exc:
```

```
617:             return [(0, f"[unreadable: {exc}]")]
618:
619:         try:
620:             tree = ast.parse(content, filename=str(py_file))
621:         except SyntaxError as exc:
622:             line = exc.lineno or 0
623:             line_text = (
624:                 content.splitlines()[line - 1].strip()
625:                 if line and line - 1 < len(content.splitlines())
626:                 else exc.msg
627:             )
628:             return [(line, f"[syntax error] {line_text}")]
629:
630:         lines = content.splitlines()
631:         for node in ast.walk(tree):
632:             if isinstance(node, ast.ImportFrom) and node.level and node.level > 0:
633:                 yield node.lineno, lines[node.lineno - 1].strip()
634:
635:     def _find_repo_root(self, package_path: Path) -> Path:
636:         """Find repository root (directory containing pyproject.toml)."""
637:         for candidate in [package_path, *package_path.parents]:
638:             if (candidate / "pyproject.toml").exists():
639:                 return candidate
640:         return package_path
641:
642:
643:
644: ================================================================================
645: FILE: src/farfan_pipeline/observability/opentelemetry_integration.py
646: ================================================================================
647:
648: """
649: FARFAN Mechanistic Policy Pipeline - OpenTelemetry Integration
650: ================================================================
651:
652: Provides comprehensive observability through OpenTelemetry spans, metrics, and traces.
653:
654: â\234\205 AUDIT_VERIFIED: Enhanced observability with OpenTelemetry spans
655:
656: Features:
657: - Distributed tracing across all 30 executors
658: - Automatic span creation and management
659: - Performance metrics collection
660: - Context propagation
661: - Integration with event tracking system
662:
663: References:
664: - OpenTelemetry Specification: https://opentelemetry.io/docs/specs/otel/
665: - Python SDK: https://opentelemetry-python.readthedocs.io/
666:
667: Author: FARFAN Team
668: Date: 2025-11-13
669: Version: 1.0.0
670: """
671:
672: import logging
```

```
import traceback
from contextlib import contextmanager
from dataclasses import dataclass
from datetime import datetime
from enum import Enum
from typing import Any

logger = logging.getLogger(__name__)


class SpanKind(Enum):
    """OpenTelemetry span kind."""
    INTERNAL = "internal"
    SERVER = "server"
    CLIENT = "client"
    PRODUCER = "producer"
    CONSUMER = "consumer"


class SpanStatus(Enum):
    """OpenTelemetry span status."""
    UNSET = "unset"
    OK = "ok"
    ERROR = "error"


@dataclass
class SpanContext:
    """
    Represents a span context for distributed tracing.

    â\234\205 AUDIT_VERIFIED: OpenTelemetry span context
    """
    trace_id: str
    span_id: str
    parent_span_id: str | None = None
    trace_flags: int = 1
    trace_state: dict[str, str] = None

    def __post_init__(self):
        if self.trace_state is None:
            self.trace_state = {}


@dataclass
class Span:
    """
    Represents an OpenTelemetry span.

    â\234\205 AUDIT_VERIFIED: Full OpenTelemetry span implementation
    """
    name: str
    context: SpanContext
    kind: SpanKind = SpanKind.INTERNAL
    status: SpanStatus = SpanStatus.UNSET
    start_time: datetime = None
```

```
    end_time: datetime | None = None
    attributes: dict[str, Any] = None
    events: list[dict[str, Any]] = None
    links: list[SpanContext] = None

    def __post_init__(self):
        if self.start_time is None:
            self.start_time = datetime.utcnow()
        if self.attributes is None:
            self.attributes = {}
        if self.events is None:
            self.events = []
        if self.links is None:
            self.links = []

    @property
    def is_recording(self) -> bool:
        """Check if span is still recording."""
        return self.end_time is None

    @property
    def duration_ms(self) -> float | None:
        """Get span duration in milliseconds."""
        if self.end_time:
            delta = self.end_time - self.start_time
            return delta.total_seconds() * 1000
        return None

    def set_attribute(self, key: str, value: Any) -> None:
        """
        Set a span attribute.

        Args:
            key: Attribute key
            value: Attribute value
        """
        if self.is_recording:
            self.attributes[key] = value

    def add_event(self, name: str, attributes: dict[str, Any] | None = None) -> None:
        """
        Add an event to the span.

        Args:
            name: Event name
            attributes: Event attributes
        """
        if self.is_recording:
            event = {
                "name": name,
                "timestamp": datetime.utcnow().isoformat(),
                "attributes": attributes or {}
            }
            self.events.append(event)

    def set_status(self, status: SpanStatus, description: str | None = None) -> None:
```

```
785:         """
786:         Set span status.
787:
788:         Args:
789:             status: Span status
790:             description: Optional status description
791:         """
792:         self.status = status
793:         if description:
794:             self.attributes["status.description"] = description
795:
796:     def record_exception(self, exception: Exception) -> None:
797:         """
798:         Record an exception in the span.
799:
800:         Args:
801:             exception: Exception to record
802:         """
803:         if self.is_recording:
804:             self.set_status(SpanStatus.ERROR, str(exception))
805:             self.add_event(
806:                 "exception",
807:                 {
808:                     "exception.type": type(exception).__name__,
809:                     "exception.message": str(exception),
810:                     "exception.stacktrace": ''.join(traceback.format_tb(exception.__traceback__))
811:                 }
812:             )
813:
814:     def end(self) -> None:
815:         """End the span."""
816:         if self.is_recording:
817:             self.end_time = datetime.utcnow()
818:
819:     def to_dict(self) -> dict[str, Any]:
820:         """Convert span to dictionary."""
821:         return {
822:             "name": self.name,
823:             "trace_id": self.context.trace_id,
824:             "span_id": self.context.span_id,
825:             "parent_span_id": self.context.parent_span_id,
826:             "kind": self.kind.value,
827:             "status": self.status.value,
828:             "start_time": self.start_time.isoformat(),
829:             "end_time": self.end_time.isoformat() if self.end_time else None,
830:             "duration_ms": self.duration_ms,
831:             "attributes": self.attributes,
832:             "events": self.events,
833:             "links": [
834:                 {"trace_id": link.trace_id, "span_id": link.span_id}
835:                 for link in self.links
836:             ]
837:         }
838:
839:
840: class Tracer:
```

```
    """
    OpenTelemetry tracer for creating and managing spans.

    â\234\205 AUDIT_VERIFIED: Tracer with automatic span management
    """

    def __init__(self, name: str, version: str = "1.0.0") -> None:
        """
        Initialize tracer.

        Args:
            name: Tracer name (usually module or component name)
            version: Tracer version
        """
        self.name = name
        self.version = version
        self.spans: list[Span] = []
        self.current_span: Span | None = None

    def start_span(
        self,
        name: str,
        kind: SpanKind = SpanKind.INTERNAL,
        attributes: dict[str, Any] | None = None,
        parent_context: SpanContext | None = None
    ) -> Span:
        """
        Start a new span.

        Args:
            name: Span name
            kind: Span kind
            attributes: Initial span attributes
            parent_context: Parent span context

        Returns:
            Started span
        """
        import uuid

        # Create span context
        trace_id = parent_context.trace_id if parent_context else str(uuid.uuid4())
        span_id = str(uuid.uuid4())
        parent_span_id = parent_context.span_id if parent_context else None

        context = SpanContext(
            trace_id=trace_id,
            span_id=span_id,
            parent_span_id=parent_span_id
        )

        # Create span
        span = Span(
            name=name,
            context=context,
            kind=kind,
```

```
            attributes=attributes or {}
        )

        # Add tracer attributes
        span.set_attribute("service.name", self.name)
        span.set_attribute("service.version", self.version)

        # Track span
        self.spans.append(span)

        logger.debug(f"Started span: {name} (trace_id={trace_id}, span_id={span_id})")

        return span

    def end_span(self, span: Span) -> None:
        """
        End a span.

        Args:
            span: Span to end
        """
        span.end()
        logger.debug(f"Ended span: {span.name} (duration={span.duration_ms:.2f}ms)")

    @contextmanager
    def start_as_current_span(
        self,
        name: str,
        kind: SpanKind = SpanKind.INTERNAL,
        attributes: dict[str, Any] | None = None
    ):
        """
        Context manager for creating a span as the current span.

        Args:
            name: Span name
            kind: Span kind
            attributes: Initial span attributes

        Yields:
            Started span

        Usage:
            >>> with tracer.start_as_current_span("operation") as span:
            ...     span.set_attribute("key", "value")
            ...     # Do work
        """
        # Get parent context from current span
        parent_context = self.current_span.context if self.current_span else None

        # Start new span
        span = self.start_span(name, kind, attributes, parent_context)

        # Set as current
        previous_span = self.current_span
        self.current_span = span
```

```
953:
954:         try:
955:             yield span
956:         except Exception as e:
957:             span.record_exception(e)
958:             raise
959:         finally:
960:             # End span
961:             self.end_span(span)
962:
963:             # Restore previous current span
964:             self.current_span = previous_span
965:
966:     def get_spans(self, trace_id: str | None = None) -> list[Span]:
967:         """
968:         Get spans, optionally filtered by trace ID.
969:
970:         Args:
971:             trace_id: Optional trace ID filter
972:
973:         Returns:
974:             List of spans
975:         """
976:         if trace_id:
977:             return [s for s in self.spans if s.context.trace_id == trace_id]
978:         return self.spans
979:
980:     def export_spans(self) -> list[dict[str, Any]]:
981:         """
982:         Export spans to dictionary format.
983:
984:         Returns:
985:             List of span dictionaries
986:         """
987:         return [span.to_dict() for span in self.spans]
988:
989:
990: class ExecutorSpanDecorator:
991:     """
992:     Decorator for automatically creating spans around executor methods.
993:
994:     â\234\205 AUDIT_VERIFIED: Automatic span creation for all 30 executors
995:
996:     Usage:
997:         >>> @executor_span("D1Q1_Executor.execute")
998:         ... def execute(self, input_data):
999:         ...     # Method implementation
1000:         ...     pass
1001:     """
1002:
1003:     def __init__(self, tracer: Tracer) -> None:
1004:         """
1005:         Initialize decorator.
1006:
1007:         Args:
1008:             tracer: Tracer to use for span creation
```

```
        """
        self.tracer = tracer

    def __call__(self, span_name: str | None = None):
        """
        Create decorator function.

        Args:
            span_name: Optional custom span name

        Returns:
            Decorator function
        """
        def decorator(func):
            def wrapper(*args, **kwargs):
                # Determine span name
                name = span_name or f"{func.__module__}.{func.__qualname__}"

                # Create attributes from function arguments
                attributes = {
                    "code.function": func.__name__,
                    "code.module": func.__module__
                }

                # Add executor-specific attributes
                if args and hasattr(args[0], "__class__"):
                    obj = args[0]
                    attributes["executor.class"] = obj.__class__.__name__

                # Start span
                with self.tracer.start_as_current_span(
                    name,
                    kind=SpanKind.INTERNAL,
                    attributes=attributes
                ) as span:
                    # Add input metadata
                    span.add_event("execution_started", {"args_count": len(args)})

                    # Execute function
                    try:
                        result = func(*args, **kwargs)

                        # Mark as successful
                        span.set_status(SpanStatus.OK)
                        span.add_event("execution_completed")

                        return result

                    except Exception as e:
                        # Record exception
                        span.record_exception(e)
                        span.add_event("execution_failed", {"error": str(e)})
                        raise

            return wrapper
        return decorator
```

```
1065:
1066:
1067: class OpenTelemetryObservability:
1068:     """
1069:     Central observability system using OpenTelemetry.
1070:
1071:     â\234\205 AUDIT_VERIFIED: Comprehensive observability with OpenTelemetry
1072:
1073:     Features:
1074:     - Distributed tracing across all executors
1075:     - Performance metrics collection
1076:     - Automatic context propagation
1077:     - Integration with existing event tracking
1078:
1079:     Usage:
1080:         >>> observability = OpenTelemetryObservability("FARFAN Pipeline")
1081:         >>> tracer = observability.get_tracer("executors")
1082:         >>> with tracer.start_as_current_span("D1Q1_execution"):
1083:         ...     # Execute D1Q1
1084:         ...     pass
1085:     """
1086:
1087:     def __init__(self, service_name: str = "farfan-pipeline", service_version: str = "1.0.0") -> None:
1088:         """
1089:         Initialize observability system.
1090:
1091:         Args:
1092:             service_name: Service name
1093:             service_version: Service version
1094:         """
1095:         self.service_name = service_name
1096:         self.service_version = service_version
1097:         self.tracers: dict[str, Tracer] = {}
1098:
1099:     def get_tracer(self, name: str) -> Tracer:
1100:         """
1101:         Get or create a tracer.
1102:
1103:         Args:
1104:             name: Tracer name
1105:
1106:         Returns:
1107:             Tracer instance
1108:         """
1109:         if name not in self.tracers:
1110:             self.tracers[name] = Tracer(name, self.service_version)
1111:             logger.info(f"Created tracer: {name}")
1112:
1113:         return self.tracers[name]
1114:
1115:     def get_executor_decorator(self, tracer_name: str = "executors") -> ExecutorSpanDecorator:
1116:         """
1117:         Get decorator for executor methods.
1118:
1119:         Args:
1120:             tracer_name: Tracer name to use
```

```
1121:
1122:         Returns:
1123:             ExecutorSpanDecorator instance
1124:         """
1125:         tracer = self.get_tracer(tracer_name)
1126:         return ExecutorSpanDecorator(tracer)
1127:
1128:     def export_all_spans(self) -> dict[str, list[dict[str, Any]]]:
1129:         """
1130:         Export all spans from all tracers.
1131:
1132:         Returns:
1133:             Dictionary mapping tracer names to span lists
1134:         """
1135:         return {
1136:             name: tracer.export_spans()
1137:             for name, tracer in self.tracers.items()
1138:         }
1139:
1140:     def get_statistics(self) -> dict[str, Any]:
1141:         """
1142:         Get observability statistics.
1143:
1144:         Returns:
1145:             Dictionary with statistics
1146:         """
1147:         total_spans = sum(len(tracer.spans) for tracer in self.tracers.values())
1148:
1149:         # Calculate average durations by tracer
1150:         tracer_stats = {}
1151:         for name, tracer in self.tracers.items():
1152:             durations = [s.duration_ms for s in tracer.spans if s.duration_ms]
1153:             tracer_stats[name] = {
1154:                 "total_spans": len(tracer.spans),
1155:                 "avg_duration_ms": sum(durations) / len(durations) if durations else 0,
1156:                 "min_duration_ms": min(durations) if durations else 0,
1157:                 "max_duration_ms": max(durations) if durations else 0
1158:             }
1159:
1160:         return {
1161:             "service_name": self.service_name,
1162:             "service_version": self.service_version,
1163:             "total_tracers": len(self.tracers),
1164:             "total_spans": total_spans,
1165:             "tracers": tracer_stats
1166:         }
1167:
1168:     def print_summary(self) -> None:
1169:         """Print observability summary."""
1170:         stats = self.get_statistics()
1171:
1172:         print("\n" + "=" * 80)
1173:         print("ð\237\224\215 OPENTELEMETRY OBSERVABILITY SUMMARY")
1174:         print("=" * 80)
1175:         print(f"Service: {stats['service_name']} v{stats['service_version']}")
1176:         print(f"Total Tracers: {stats['total_tracers']}")
```

```
        print(f"Total Spans: {stats['total_spans']}")

        if stats['tracers']:
            print("\n" + "-" * 80)
            print("Tracer Statistics:")
            print("-" * 80)

            for tracer_name, tracer_stats in stats['tracers'].items():
                print(f"\n{tracer_name}:")
                print(f"  Total Spans: {tracer_stats['total_spans']}")
                print(f"  Avg Duration: {tracer_stats['avg_duration_ms']:.2f}ms")
                print(f"  Min Duration: {tracer_stats['min_duration_ms']:.2f}ms")
                print(f"  Max Duration: {tracer_stats['max_duration_ms']:.2f}ms")

        print("\n" + "=" * 80)


# Global observability instance
_global_observability: OpenTelemetryObservability | None = None


def get_global_observability() -> OpenTelemetryObservability:
    """Get or create global observability instance."""
    global _global_observability
    if _global_observability is None:
        _global_observability = OpenTelemetryObservability("FARFAN Pipeline", "1.0.0")
    return _global_observability


def get_tracer(name: str) -> Tracer:
    """Convenience function to get tracer from global observability."""
    return get_global_observability().get_tracer(name)


def executor_span(span_name: str | None = None):
    """
    Convenience decorator for executor methods.

    Usage:
        >>> @executor_span("D1Q1_Executor.execute")
        ... def execute(self, input_data):
        ...     pass
    """
    observability = get_global_observability()
    decorator = observability.get_executor_decorator()
    return decorator(span_name)


# â\234\205 AUDIT_VERIFIED: OpenTelemetry Integration Complete
# - Distributed tracing with full span support
# - Automatic span creation for executors
# - Performance metrics collection
# - Context propagation
# - Integration-ready with existing systems


```

```
1233:
1234: ================================================================================
1235: FILE: src/farfan_pipeline/observability/path_guard.py
1236: ================================================================================
1237:
1238: """
1239: Path Guard
1240:
1241: Runtime interception of filesystem operations.
1242: Validates paths against PathPolicy during execution.
1243:
1244: NOTE:
1245: This module is verification-focused: it logs violations into PolicyReport.
1246: Whether to treat violations as hard failures is decided by the harness.
1247: """
1248:
1249: from __future__ import annotations
1250:
1251: import builtins
1252: import os
1253: from contextlib import contextmanager
1254: from pathlib import Path
1255:
1256: from farfan_pipeline.observability.path_import_policy import ImportPolicy, PathPolicy, PolicyReport, PolicyViolation
1257:
1258:
1259: def _is_under_prefixes(resolved: Path, prefixes: frozenset[Path]) -> bool:
1260:     """Return True if 'resolved' is under any prefix in 'prefixes'."""
1261:     for prefix in prefixes:
1262:         try:
1263:             resolved.relative_to(prefix)
1264:             return True
1265:         except ValueError:
1266:             continue
1267:     return False
1268:
1269:
1270: def _path_allowed(
1271:     path: Path,
1272:     operation: str,
1273:     policy: PathPolicy,
1274: ) -> tuple[bool, str | None]:
1275:     """
1276:     Check if path operation is allowed by policy.
1277:
1278:     Args:
1279:         path: Path being accessed.
1280:         operation: "read" or "write".
1281:         policy: PathPolicy to validate against.
1282:
1283:     Returns:
1284:         (allowed, error_message)
1285:     """
1286:     try:
1287:         resolved = path.resolve()
1288:     except Exception as e:
```

```
1289:         return False, f"Cannot resolve path {path!r}: {e}"
1290:
1291:     if operation == "write":
1292:         if _is_under_prefixes(resolved, policy.allowed_write_prefixes) or _is_under_prefixes(
1293:             resolved, policy.allowed_external_prefixes
1294:         ):
1295:             return True, None
1296:         return False, f"Write not allowed: {resolved} not under allowed write prefixes"
1297:
1298:     if operation == "read":
1299:         if _is_under_prefixes(resolved, policy.allowed_read_prefixes) or _is_under_prefixes(
1300:             resolved, policy.allowed_external_prefixes
1301:         ):
1302:             return True, None
1303:         return False, f"Read not allowed: {resolved} not under allowed read prefixes"
1304:
1305:     return True, None
1306:
1307:
1308: @contextmanager
1309: def guard_paths_and_imports(
1310:     path_policy: PathPolicy,
1311:     import_policy: ImportPolicy,  # kept for future dynamic-import guards
1312:     report: PolicyReport,
1313: ):
1314:     """
1315:     Context manager that guards filesystem paths during execution.
1316:
1317:     Args:
1318:         path_policy: Policy for filesystem access.
1319:         import_policy: Policy for imports (reserved for future dynamic import guards).
1320:         report: PolicyReport to populate with violations.
1321:
1322:     Note:
1323:         This implementation logs violations but does not block execution by itself.
1324:         The harness decides whether any violation is fatal.
1325:     """
1326:     original_open = builtins.open
1327:     original_os_open = os.open
1328:
1329:     in_guard = {"value": False}
1330:
1331:     def guarded_open(file, mode="r", *args, **kwargs):
1332:         if in_guard["value"]:
1333:             return original_open(file, mode, *args, **kwargs)
1334:
1335:         in_guard["value"] = True
1336:         try:
1337:             path = Path(file)
1338:             is_write = any(m in mode for m in ("w", "a", "x", "+"))
1339:             operation = "write" if is_write else "read"
1340:
1341:             allowed, error_msg = _path_allowed(path, operation, path_policy)
1342:             if not allowed:
1343:                 report.path_violations.append(
1344:                     PolicyViolation(
```

```
                        kind="path",
                        message=error_msg or f"Path access violation: {operation}",
                        file=path,
                        line=None,
                        operation=operation,
                        target=str(path),
                    )
                )

            return original_open(file, mode, *args, **kwargs)
        finally:
            in_guard["value"] = False

    def guarded_os_open(path, flags, *args, **kwargs):
        if in_guard["value"]:
            return original_os_open(path, flags, *args, **kwargs)

        in_guard["value"] = True
        try:
            path_obj = Path(path)
            is_write = bool(
                flags & (os.O_WRONLY | os.O_RDWR | os.O_CREAT | os.O_TRUNC | os.O_APPEND)
            )
            operation = "write" if is_write else "read"

            allowed, error_msg = _path_allowed(path_obj, operation, path_policy)
            if not allowed:
                report.path_violations.append(
                    PolicyViolation(
                        kind="path",
                        message=error_msg or f"Path access violation: {operation}",
                        file=path_obj,
                        line=None,
                        operation=operation,
                        target=str(path_obj),
                    )
                )

            return original_os_open(path, flags, *args, **kwargs)
        finally:
            in_guard["value"] = False

    builtins.open = guarded_open
    os.open = guarded_os_open

    try:
        yield
    finally:
        builtins.open = original_open
        os.open = original_os_open


================================================================================
FILE: src/farfan_pipeline/observability/path_import_policy.py
================================================================================

```

```
1401: """
1402: Path and Import Policy Contracts
1403:
1404: Authoritative, type-safe policy definitions for validating imports and filesystem paths.
1405: Used by both static analysis (AST) and dynamic runtime guards.
1406:
1407: This module is the single source of truth for:
1408: - How violations are represented.
1409: - How success/failure is computed.
1410: - How reports are serialized into verification_manifest.json.
1411: """
1412:
1413: from __future__ import annotations
1414:
1415: from dataclasses import dataclass, field
1416: from pathlib import Path
1417: from typing import FrozenSet, List, Literal, Optional, Dict, Any
1418:
1419:
1420: PolicyKind = Literal["static_import", "dynamic_import", "path", "sys_path"]
1421:
1422:
1423: @dataclass(frozen=True)
1424: class PolicyViolation:
1425:     """A single policy violation detected during verification."""
1426:
1427:     kind: PolicyKind
1428:     message: str
1429:     file: Optional[Path] = None
1430:     line: Optional[int] = None
1431:     operation: Optional[str] = None  # e.g. "open", "import", "rename"
1432:     target: Optional[str] = None  # module name or path string
1433:
1434:     def to_dict(self) -> Dict[str, Any]:
1435:         """Convert to dictionary for JSON serialization."""
1436:         return {
1437:             "kind": self.kind,
1438:             "message": self.message,
1439:             "file": str(self.file) if self.file else None,
1440:             "line": self.line,
1441:             "operation": self.operation,
1442:             "target": self.target,
1443:         }
1444:
1445:
1446: @dataclass
1447: class PolicyReport:
1448:     """
1449:     Aggregated report of all policy violations.
1450:
1451:     A run is considered "clean" only if violation_count() == 0.
1452:     """
1453:
1454:     static_import_violations: List[PolicyViolation] = field(default_factory=list)
1455:     dynamic_import_violations: List[PolicyViolation] = field(default_factory=list)
1456:     path_violations: List[PolicyViolation] = field(default_factory=list)
```

```
    sys_path_violations: List[PolicyViolation] = field(default_factory=list)

    def ok(self) -> bool:
        """Return True if no violations exist."""
        return self.violation_count() == 0

    def violation_count(self) -> int:
        """Total number of violations."""
        return (
            len(self.static_import_violations)
            + len(self.dynamic_import_violations)
            + len(self.path_violations)
            + len(self.sys_path_violations)
        )

    def to_dict(self) -> Dict[str, Any]:
        """
        Convert to dictionary form, suitable for embedding in verification_manifest.json.
        """
        return {
            "success": self.ok(),
            "violation_count": self.violation_count(),
            "static_import_violations": [v.to_dict() for v in self.static_import_violations],
            "dynamic_import_violations": [v.to_dict() for v in self.dynamic_import_violations],
            "path_violations": [v.to_dict() for v in self.path_violations],
            "sys_path_violations": [v.to_dict() for v in self.sys_path_violations],
        }


@dataclass(frozen=True)
class ImportPolicy:
    """
    Policy controlling allowed imports.

    "Maximum hardness" interpretation:
    - Only modules that match:
        - allowed_internal_prefixes, or
        - stdlib_modules, or
        - allowed_third_party
      are allowed.
    - Dynamic imports must use module names in allowed_dynamic_imports.
    """

    allowed_internal_prefixes: FrozenSet[str]
    allowed_third_party: FrozenSet[str]
    allowed_dynamic_imports: FrozenSet[str]
    stdlib_modules: FrozenSet[str]


@dataclass(frozen=True)
class PathPolicy:
    """
    Policy controlling filesystem access.

    "Maximum hardness" interpretation:
    - Any path access outside the cones encoded here is a violation.
```

```
    """

    repo_root: Path
    allowed_write_prefixes: FrozenSet[Path]
    allowed_read_prefixes: FrozenSet[Path]
    allowed_external_prefixes: FrozenSet[Path]


def merge_policy_reports(reports: List[PolicyReport]) -> PolicyReport:
    """
    Merge multiple PolicyReport objects into one.

    Args:
        reports: List of PolicyReport objects.

    Returns:
        Merged PolicyReport with all violations.
    """
    merged = PolicyReport()

    for report in reports:
        merged.static_import_violations.extend(report.static_import_violations)
        merged.dynamic_import_violations.extend(report.dynamic_import_violations)
        merged.path_violations.extend(report.path_violations)
        merged.sys_path_violations.extend(report.sys_path_violations)

    return merged


================================================================================
FILE: src/farfan_pipeline/observability/policy_builder.py
================================================================================

"""
Policy Builder

Constructs ImportPolicy and PathPolicy from the actual repository state.
Anchored to real files and dependencies.

"Maximum hardness" assumptions:
- Repo root MUST contain farfan_core/farfan_core.
- Third-party allowlist MUST come from an explicit dependency_lockdown module.
- Any failure to compute a clear policy is treated as a fatal configuration error.
"""

from __future__ import annotations

import site
import sys
import tempfile
from pathlib import Path
from typing import FrozenSet, Set

from farfan_pipeline.observability.path_import_policy import ImportPolicy, PathPolicy


```

```
1569: def compute_repo_root() -> Path:
1570:     """
1571:     Compute repository root by locating the directory containing farfan_core package.
1572:
1573:     Strategy:
1574:     - Walk parents of this file until we find one that contains farfan_core/farfan_core.
1575:     - Fallback to farfan_core.config.paths.PROJECT_ROOT if available.
1576:     - If nothing matches, fail hard.
1577:
1578:     Returns:
1579:         Path to repository root.
1580:
1581:     Raises:
1582:         RuntimeError: If repo root cannot be determined.
1583:     """
1584:     here = Path(__file__).resolve()
1585:
1586:     for candidate in here.parents:
1587:         if (candidate / "farfan_core" / "farfan_core").exists():
1588:             return candidate
1589:
1590:     # Fallback: use PROJECT_ROOT from config if available
1591:     try:
1592:         from farfan_pipeline.config.paths import PROJECT_ROOT  # type: ignore[attr-defined]
1593:
1594:         project_root = Path(PROJECT_ROOT).resolve()
1595:         if (project_root / "farfan_core" / "farfan_core").exists():
1596:             return project_root
1597:     except Exception:
1598:         pass
1599:
1600:     raise RuntimeError(
1601:         f"Cannot determine repo root. Scanned parents of {here} and did not find a "
1602:         f"directory containing farfan_core/farfan_core, nor a valid PROJECT_ROOT."
1603:     )
1604:
1605:
1606: def _load_third_party_from_lockdown(repo_root: Path) -> FrozenSet[str]:
1607:     """
1608:     Load third-party module allowlist from an explicit lockdown module.
1609:
1610:     Maximum hardness:
1611:     - We do NOT silently fall back to requirements.txt.
1612:     - If the lockdown module is missing or empty, this is a configuration error.
1613:     """
1614:     try:
1615:         # Module is expected to live under farfan_core/farfan_core/config/dependency_lockdown.py
1616:         from farfan_pipeline.config.dependency_lockdown import ALLOWED_THIRD_PARTY_MODULES  # type: ignore[attr-defined]
1617:
1618:         allowed = frozenset(ALLOWED_THIRD_PARTY_MODULES)
1619:         if not allowed:
1620:             raise RuntimeError("ALLOWED_THIRD_PARTY_MODULES is empty; lockdown too weak.")
1621:         return allowed
1622:     except Exception as exc:
1623:         raise RuntimeError(
1624:             "Failed to load ALLOWED_THIRD_PARTY_MODULES from "
```

```
            "farfan_core.config.dependency_lockdown. This module MUST exist "
            "and define a non-empty ALLOWED_THIRD_PARTY_MODULES FrozenSet[str]."
        ) from exc


def build_import_policy(repo_root: Path) -> ImportPolicy:
    """
    Build ImportPolicy from repository state.

    Args:
        repo_root: Repository root path.

    Returns:
        ImportPolicy with allowed imports.

    Maximum hardness:
    - Internal imports allowed only under 'farfan_core.*'.
    - Third-party imports allowed only if explicitly declared in dependency_lockdown.
    - Dynamic imports allowed only if explicitly declared in the same lockdown file (optional).
    """
    # Internal prefixes: explicit farfan_core namespace.
    allowed_internal: FrozenSet[str] = frozenset(
        {
            "farfan_core.core",
            "farfan_core.entrypoint",
            "farfan_core.scripts",
            "farfan_core.analysis",
            "farfan_core.audit",
            "farfan_core.observability",
            "farfan_core.utils",
            "farfan_core.scoring",
            "farfan_core.processing",
            "farfan_core.patterns",
            "farfan_core.infrastructure",
            "farfan_core.flux",
            "farfan_core.config",
            "farfan_core.controls",
            "farfan_core.concurrency",
            "farfan_core.optimization",
            "farfan_core.api",
            "farfan_core",
        }
    )

    # Stdlib modules set
    stdlib: FrozenSet[str] = (
        frozenset(sys.stdlib_module_names)
        if hasattr(sys, "stdlib_module_names")
        else frozenset()
    )

    # Third-party lockdown (MUST exist)
    allowed_third_party: FrozenSet[str] = _load_third_party_from_lockdown(repo_root)

    # Dynamic imports: default empty, but can be extended in lockdown config
    allowed_dynamic: Set[str] = set()
```

```
    try:
        from farfan_pipeline.config.dependency_lockdown import ALLOWED_DYNAMIC_IMPORTS  # type: ignore[attr-defined]

        allowed_dynamic.update(ALLOWED_DYNAMIC_IMPORTS)
    except Exception:
        # Optional; absence means "no dynamic imports whitelisted"
        pass

    return ImportPolicy(
        allowed_internal_prefixes=allowed_internal,
        allowed_third_party=allowed_third_party,
        allowed_dynamic_imports=frozenset(allowed_dynamic),
        stdlib_modules=stdlib,
    )


def build_path_policy(repo_root: Path) -> PathPolicy:
    """
    Build PathPolicy from repository state.

    Args:
        repo_root: Repository root path.

    Returns:
        PathPolicy with allowed paths.

    Maximum hardness:
    - Writes allowed only under known artifact/log/output dirs (or explicit external prefixes).
    - Reads allowed only under repo_root or explicit external prefixes.
    """
    # Allowed write prefixes (where artifacts / logs / outputs can be written)
    allowed_write: FrozenSet[Path] = frozenset(
        {
            repo_root / "artifacts",
            repo_root / "logs",
            repo_root / "output",
        }
    )

    # Allowed read prefixes: entire repo
    allowed_read: FrozenSet[Path] = frozenset(
        {
            repo_root,
        }
    )

    # Allowed external prefixes (outside repo)
    external_paths: Set[Path] = set()

    # Temp directory
    external_paths.add(Path(tempfile.gettempdir()))

    # Site-packages (installed dependencies)
    try:
        for site_dir in site.getsitepackages():
            external_paths.add(Path(site_dir))
```

```
    except Exception:
        # getsitepackages may not exist in some environments (e.g. venvs)
        pass

    # User site-packages
    try:
        if site.ENABLE_USER_SITE:
            user_site = site.getusersitepackages()
            if user_site:
                external_paths.add(Path(user_site))
    except Exception:
        pass

    allowed_external: FrozenSet[Path] = frozenset(external_paths)

    return PathPolicy(
        repo_root=repo_root,
        allowed_write_prefixes=allowed_write,
        allowed_read_prefixes=allowed_read,
        allowed_external_prefixes=allowed_external,
    )


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

"""
FARFAN Mechanistic Policy Pipeline - Optimization Module
=========================================================

Reinforcement learning-based optimization for continuous improvement
of execution strategies.

Author: FARFAN Team
Date: 2025-11-13
Version: 1.0.0
"""

from farfan_pipeline.optimization.rl_strategy import (
    BanditAlgorithm,
    BanditArm,
    EpsilonGreedyAlgorithm,
    ExecutorMetrics,
    OptimizationStrategy,
    RLStrategyOptimizer,
    ThompsonSamplingAlgorithm,
    UCB1Algorithm,
)

__all__ = [
    "BanditAlgorithm",
    "BanditArm",
    "EpsilonGreedyAlgorithm",
    "ExecutorMetrics",
    "OptimizationStrategy",
```

```
    "RLStrategyOptimizer",
    "ThompsonSamplingAlgorithm",
    "UCB1Algorithm",
]



================================================================================
FILE: src/farfan_pipeline/optimization/rl_strategy.py
================================================================================

"""
FARFAN Mechanistic Policy Pipeline - RL-Based Strategy Optimization
====================================================================

Implements reinforcement learning-based optimization for continuous improvement
of executor selection and orchestration strategies.

Uses multi-armed bandit algorithms (Thompson Sampling, UCB) to learn optimal
execution strategies over time based on performance metrics.

â\234\205 AUDIT_VERIFIED: RL-based Strategy Optimization for continuous improvement

References:
- Sutton & Barto (2018): "Reinforcement Learning: An Introduction"
- Agrawal & Goyal (2012): "Analysis of Thompson Sampling for MAB"
- Auer et al. (2002): "UCB algorithms for multi-armed bandit problems"

Author: FARFAN Team
Date: 2025-11-13
Version: 1.0.0
"""

import json
import logging
import math
import random
from abc import ABC, abstractmethod
from collections import deque
from dataclasses import dataclass, field
from datetime import datetime
from enum import Enum
from pathlib import Path
from typing import Any
from uuid import uuid4

import numpy as np
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method

logger = logging.getLogger(__name__)


class OptimizationStrategy(Enum):
    """RL optimization strategy."""
    THOMPSON_SAMPLING = "thompson_sampling"  # Bayesian approach
```

```
    UCB1 = "ucb1"  # Upper Confidence Bound
    EPSILON_GREEDY = "epsilon_greedy"  # Simple exploration-exploitation
    EXP3 = "exp3"  # Exponential-weight algorithm for exploration and exploitation


@dataclass
class ExecutorMetrics:
    """
    Metrics for a single executor execution.

    â\234\205 AUDIT_VERIFIED: Comprehensive performance tracking
    """
    executor_name: str
    execution_id: str = field(default_factory=lambda: str(uuid4()))
    timestamp: datetime = field(default_factory=datetime.utcnow)
    success: bool = False
    duration_ms: float = 0.0
    quality_score: float = 0.0  # 0.0 to 1.0
    tokens_used: int = 0
    cost_usd: float = 0.0
    error: str | None = None
    metadata: dict[str, Any] = field(default_factory=dict)

    @property
    @calibrated_method("farfan_core.optimization.rl_strategy.ExecutorMetrics.reward")
    def reward(self) -> float:
        """
        Calculate reward for RL algorithm.

        Reward combines:
        - Success (binary)
        - Quality score (0-1)
        - Efficiency (inverse of duration, normalized)
        - Cost efficiency (inverse of cost, normalized)

        Returns:
            Normalized reward between 0 and 1
        """
        if not self.success:
            return ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.ExecutorMetrics.reward", "auto_param_L85_19", 0.0)

        # Base reward from quality
        quality_reward = self.quality_score

        # Efficiency reward (faster is better, normalized to 0-1)
        # Assume typical execution is 1000ms, scale accordingly
        typical_duration = 1000.0
        efficiency_reward = max(0, 1 - (self.duration_ms / (2 * typical_duration)))

        # Cost efficiency reward (cheaper is better, normalized to 0-1)
        # Assume typical cost is $ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.ExecutorMetrics.reward", "auto_param_L96_34", 0.01), scale acc
ordingly
        typical_cost = ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.ExecutorMetrics.reward", "typical_cost", 0.01) # Refactored
        cost_reward = max(0, 1 - (self.cost_usd / (2 * typical_cost)))

        # Weighted combination
```

```
1904:         reward = (
1905:             ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.ExecutorMetrics.reward", "auto_param_L102_12", 0.5) * quality_reward +
1906:             ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.ExecutorMetrics.reward", "auto_param_L103_12", 0.3) * efficiency_reward +
1907:             ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.ExecutorMetrics.reward", "auto_param_L104_12", 0.2) * cost_reward
1908:         )
1909:
1910:         return min(ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.ExecutorMetrics.reward", "auto_param_L107_19", 1.0), max(ParameterLoaderV2.ge
t("farfan_core.optimization.rl_strategy.ExecutorMetrics.reward", "auto_param_L107_28", 0.0), reward))
1911:
1912:
1913: @dataclass
1914: class BanditArm:
1915:     """
1916:     Represents a bandit arm (executor or strategy choice).
1917:
1918:     â\234\205 AUDIT_VERIFIED: Bayesian posterior tracking for Thompson Sampling
1919:     """
1920:     arm_id: str
1921:     name: str
1922:
1923:     # Bayesian posterior (Beta distribution for Thompson Sampling)
1924:     alpha: float = ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.ExecutorMetrics.reward", "auto_param_L121_19", 1.0)  # Successes + 1
1925:     beta: float = ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.ExecutorMetrics.reward", "auto_param_L122_18", 1.0)   # Failures + 1
1926:
1927:     # Empirical statistics
1928:     pulls: int = 0
1929:     total_reward: float = ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.ExecutorMetrics.reward", "auto_param_L126_26", 0.0)
1930:     successes: int = 0
1931:     failures: int = 0
1932:
1933:     # Performance tracking
1934:     total_duration_ms: float = ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.ExecutorMetrics.reward", "auto_param_L131_31", 0.0)
1935:     total_tokens: int = 0
1936:     total_cost_usd: float = ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.ExecutorMetrics.reward", "auto_param_L133_28", 0.0)
1937:
1938:     # Recent performance (last N executions)
1939:     recent_rewards: deque = field(default_factory=lambda: deque(maxlen=100))
1940:     max_recent: int = 100
1941:
1942:     @property
1943:     @calibrated_method("farfan_core.optimization.rl_strategy.BanditArm.mean_reward")
1944:     def mean_reward(self) -> float:
1945:         """Calculate mean reward."""
1946:         return self.total_reward / self.pulls if self.pulls > 0 else ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.BanditArm.mean_reward", "au
to_param_L143_69", 0.0)
1947:
1948:     @property
1949:     @calibrated_method("farfan_core.optimization.rl_strategy.BanditArm.success_rate")
1950:     def success_rate(self) -> float:
1951:         """Calculate success rate."""
1952:         total = self.successes + self.failures
1953:         return self.successes / total if total > 0 else ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.BanditArm.success_rate", "auto_param_L15
0_56", 0.0)
1954:
1955:     @property
1956:     @calibrated_method("farfan_core.optimization.rl_strategy.BanditArm.mean_duration_ms")
```

```
    def mean_duration_ms(self) -> float:
        """Calculate mean duration."""
        return self.total_duration_ms / self.pulls if self.pulls > 0 else ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.BanditArm.mean_duration_ms", "auto_param_L156_74", 0.0)

    @property
    @calibrated_method("farfan_core.optimization.rl_strategy.BanditArm.mean_cost_usd")
    def mean_cost_usd(self) -> float:
        """Calculate mean cost."""
        return self.total_cost_usd / self.pulls if self.pulls > 0 else ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.BanditArm.mean_cost_usd", "auto_param_L162_71", 0.0)

    @calibrated_method("farfan_core.optimization.rl_strategy.BanditArm.update")
    def update(self, metrics: ExecutorMetrics) -> None:
        """
        Update arm statistics with new execution metrics.

        Args:
            metrics: Execution metrics
        """
        reward = metrics.reward

        # Update counts
        self.pulls += 1

        # Update Bayesian posterior
        if metrics.success and reward > ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.BanditArm.update", "auto_param_L178_40", 0.5):
            self.alpha += 1
            self.successes += 1
        else:
            self.beta += 1
            self.failures += 1

        # Update empirical statistics
        self.total_reward += reward
        self.total_duration_ms += metrics.duration_ms
        self.total_tokens += metrics.tokens_used
        self.total_cost_usd += metrics.cost_usd

        # Update recent rewards (sliding window with deque automatically handles maxlen)
        self.recent_rewards.append(reward)

    @calibrated_method("farfan_core.optimization.rl_strategy.BanditArm.sample_thompson")
    def sample_thompson(self, rng: np.random.Generator) -> float:
        """
        Sample from Thompson Sampling posterior (Beta distribution).

        Args:
            rng: NumPy random generator

        Returns:
            Sampled success probability
        """
        return rng.beta(self.alpha, self.beta)

    @calibrated_method("farfan_core.optimization.rl_strategy.BanditArm.ucb_score")
```

```
    def ucb_score(self, total_pulls: int, c: float = 2.0) -> float:
        """
        Calculate UCB1 score.

        Args:
            total_pulls: Total pulls across all arms
            c: Exploration parameter

        Returns:
            UCB score
        """
        if self.pulls == 0:
            return float('inf')

        exploitation = self.mean_reward
        exploration = c * math.sqrt(math.log(total_pulls) / self.pulls)

        return exploitation + exploration

    @calibrated_method("farfan_core.optimization.rl_strategy.BanditArm.to_dict")
    def to_dict(self) -> dict[str, Any]:
        """Convert arm to dictionary."""
        return {
            "arm_id": self.arm_id,
            "name": self.name,
            "alpha": self.alpha,
            "beta": self.beta,
            "pulls": self.pulls,
            "mean_reward": self.mean_reward,
            "success_rate": self.success_rate,
            "mean_duration_ms": self.mean_duration_ms,
            "mean_cost_usd": self.mean_cost_usd,
            "total_tokens": self.total_tokens
        }


class BanditAlgorithm(ABC):
    """Base class for bandit algorithms."""

    @abstractmethod
    @calibrated_method("farfan_core.optimization.rl_strategy.BanditAlgorithm.select_arm")
    def select_arm(self, arms: list[BanditArm], rng: np.random.Generator) -> BanditArm:
        """
        Select an arm to pull.

        Args:
            arms: Available arms
            rng: Random number generator

        Returns:
            Selected arm
        """
        pass


class ThompsonSamplingAlgorithm(BanditAlgorithm):
```

```
    """
    Thompson Sampling algorithm for bandit optimization.

    Bayesian approach that maintains posterior distributions over success
    probabilities and samples from them for exploration-exploitation balance.

    â\234\205 AUDIT_VERIFIED: Thompson Sampling implementation
    """

    @calibrated_method("farfan_core.optimization.rl_strategy.ThompsonSamplingAlgorithm.select_arm")
    def select_arm(self, arms: list[BanditArm], rng: np.random.Generator) -> BanditArm:
        """Select arm using Thompson Sampling."""
        if not arms:
            raise ValueError("No arms available")

        # Sample from each arm's posterior
        samples = [arm.sample_thompson(rng) for arm in arms]

        # Select arm with highest sample
        best_idx = int(np.argmax(samples))

        logger.debug(f"Thompson Sampling: Selected {arms[best_idx].name} (sample: {samples[best_idx]:.4f})")

        return arms[best_idx]


class UCB1Algorithm(BanditAlgorithm):
    """
    UCB1 (Upper Confidence Bound) algorithm.

    Deterministic approach that balances exploitation and exploration using
    confidence bounds.

    â\234\205 AUDIT_VERIFIED: UCB1 implementation
    """

    def __init__(self, c: float = 2.0) -> None:
        """
        Initialize UCB1 algorithm.

        Args:
            c: Exploration parameter (higher = more exploration)
        """
        self.c = c

    @calibrated_method("farfan_core.optimization.rl_strategy.UCB1Algorithm.select_arm")
    def select_arm(self, arms: list[BanditArm], rng: np.random.Generator) -> BanditArm:
        """Select arm using UCB1."""
        if not arms:
            raise ValueError("No arms available")

        # Force exploration of unplayed arms first
        unplayed = [arm for arm in arms if arm.pulls == 0]
        if unplayed:
            selected = random.choice(unplayed)
            logger.debug(f"UCB1: Exploring unplayed arm {selected.name}")
```

```
2123:             return selected
2124:
2125:         # Calculate UCB scores
2126:         total_pulls = sum(arm.pulls for arm in arms)
2127:         scores = [arm.ucb_score(total_pulls, self.c) for arm in arms]
2128:
2129:         # Select arm with highest UCB score
2130:         best_idx = int(np.argmax(scores))
2131:
2132:         logger.debug(f"UCB1: Selected {arms[best_idx].name} (UCB: {scores[best_idx]:.4f})")
2133:
2134:         return arms[best_idx]
2135:
2136:
2137: class EpsilonGreedyAlgorithm(BanditAlgorithm):
2138:     """
2139:     Epsilon-Greedy algorithm.
2140:
2141:     Simple approach: with probability epsilon, explore randomly;
2142:     otherwise, exploit best known arm.
2143:
2144:     â\234\205 AUDIT_VERIFIED: Epsilon-Greedy implementation
2145:     """
2146:
2147:     def __init__(self, epsilon: float = 0.1, decay: bool = False) -> None:
2148:         """
2149:         Initialize Epsilon-Greedy algorithm.
2150:
2151:         Args:
2152:             epsilon: Exploration probability (0-1)
2153:             decay: Whether to decay epsilon over time
2154:         """
2155:         self.epsilon = epsilon
2156:         self.initial_epsilon = epsilon
2157:         self.decay = decay
2158:         self.total_selections = 0
2159:
2160:     @calibrated_method("farfan_core.optimization.rl_strategy.EpsilonGreedyAlgorithm.select_arm")
2161:     def select_arm(self, arms: list[BanditArm], rng: np.random.Generator) -> BanditArm:
2162:         """Select arm using Epsilon-Greedy."""
2163:         if not arms:
2164:             raise ValueError("No arms available")
2165:
2166:         self.total_selections += 1
2167:
2168:         # Decay epsilon if enabled
2169:         if self.decay:
2170:             self.epsilon = self.initial_epsilon / (1 + ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.EpsilonGreedyAlgorithm.select_arm", "auto
_param_L367_55", 0.001) * self.total_selections)
2171:
2172:         # Explore with probability epsilon
2173:         if rng.random() < self.epsilon:
2174:             selected = random.choice(arms)
2175:             logger.debug(f"Epsilon-Greedy: Exploring {selected.name} (Îµ={self.epsilon:.4f})")
2176:             return selected
2177:
```

```
        # Exploit: select best arm by mean reward
        best_arm = max(arms, key=lambda a: a.mean_reward if a.pulls > 0 else 0)
        logger.debug(f"Epsilon-Greedy: Exploiting {best_arm.name} (reward={best_arm.mean_reward:.4f})")

        return best_arm


class RLStrategyOptimizer:
    """
    RL-based strategy optimizer for executor selection.

    â\234\205 AUDIT_VERIFIED: RL-based Strategy Optimization for continuous improvement

    Usage:
        >>> optimizer = RLStrategyOptimizer(
        ...     strategy=OptimizationStrategy.THOMPSON_SAMPLING,
        ...     arms=["D1Q1_Executor", "D1Q2_Executor"]
        ... )
        >>> selected = optimizer.select_executor()
        >>> metrics = ExecutorMetrics(...)
        >>> optimizer.update(selected, metrics)
    """

    def __init__(
        self,
        strategy: OptimizationStrategy = OptimizationStrategy.THOMPSON_SAMPLING,
        arms: list[str] | None = None,
        seed: int = 42
    ) -> None:
        """
        Initialize RL strategy optimizer.

        Args:
            strategy: Optimization strategy to use
            arms: List of arm names (executors or strategies)
            seed: Random seed for reproducibility
        """
        self.strategy = strategy
        self.rng = np.random.default_rng(seed)
        self.optimizer_id = str(uuid4())
        self.created_at = datetime.utcnow()

        # Initialize arms
        self.arms: dict[str, BanditArm] = {}
        if arms:
            for arm_name in arms:
                self.add_arm(arm_name)

        # Select algorithm
        self.algorithm = self._create_algorithm(strategy)

        # Execution history
        self.history: list[tuple[str, ExecutorMetrics]] = []

    @calibrated_method("farfan_core.optimization.rl_strategy.RLStrategyOptimizer._create_algorithm")
    def _create_algorithm(self, strategy: OptimizationStrategy) -> BanditAlgorithm:
```

```
2234:         """Create bandit algorithm based on strategy."""
2235:         if strategy == OptimizationStrategy.THOMPSON_SAMPLING:
2236:             return ThompsonSamplingAlgorithm()
2237:         elif strategy == OptimizationStrategy.UCB1:
2238:             return UCB1Algorithm(c=2.0)
2239:         elif strategy == OptimizationStrategy.EPSILON_GREEDY:
2240:             return EpsilonGreedyAlgorithm(epsilon=ParameterLoaderV2.get("farfan_core.optimization.rl_strategy.RLStrategyOptimizer._create_algorithm", "auto_
param_L437_50", 0.1), decay=True)
2241:         else:
2242:             raise ValueError(f"Unsupported strategy: {strategy}")
2243: 
2244:     @calibrated_method("farfan_core.optimization.rl_strategy.RLStrategyOptimizer.add_arm")
2245:     def add_arm(self, name: str) -> BanditArm:
2246:         """
2247:         Add a new arm to the optimizer.
2248: 
2249:         Args:
2250:             name: Arm name (executor or strategy)
2251: 
2252:         Returns:
2253:             Created arm
2254:         """
2255:         arm_id = f"{name}_{str(uuid4())[:8]}"
2256:         arm = BanditArm(arm_id=arm_id, name=name)
2257:         self.arms[name] = arm
2258:         logger.info(f"Added arm: {name}")
2259:         return arm
2260: 
2261:     @calibrated_method("farfan_core.optimization.rl_strategy.RLStrategyOptimizer.select_arm")
2262:     def select_arm(self) -> str:
2263:         """
2264:         Select an arm using the configured algorithm.
2265: 
2266:         Returns:
2267:             Selected arm name
2268:         """
2269:         if not self.arms:
2270:             raise ValueError("No arms configured")
2271: 
2272:         arms_list = list(self.arms.values())
2273:         selected_arm = self.algorithm.select_arm(arms_list, self.rng)
2274: 
2275:         return selected_arm.name
2276: 
2277:     @calibrated_method("farfan_core.optimization.rl_strategy.RLStrategyOptimizer.update")
2278:     def update(self, arm_name: str, metrics: ExecutorMetrics) -> None:
2279:         """
2280:         Update arm statistics with execution metrics.
2281: 
2282:         Args:
2283:             arm_name: Name of the executed arm
2284:             metrics: Execution metrics
2285: 
2286:         Raises:
2287:             ValueError: If arm not found
2288:         """
```

```
        if arm_name not in self.arms:
            raise ValueError(f"Arm not found: {arm_name}")

        arm = self.arms[arm_name]
        arm.update(metrics)

        # Add to history
        self.history.append((arm_name, metrics))

        logger.info(
            f"Updated arm {arm_name}: "
            f"pulls={arm.pulls}, "
            f"mean_reward={arm.mean_reward:.4f}, "
            f"success_rate={arm.success_rate:.2%}"
        )

    @calibrated_method("farfan_core.optimization.rl_strategy.RLStrategyOptimizer.get_statistics")
    def get_statistics(self) -> dict[str, Any]:
        """
        Get optimizer statistics.

        Returns:
            Dictionary with statistics
        """
        total_pulls = sum(arm.pulls for arm in self.arms.values())

        return {
            "optimizer_id": self.optimizer_id,
            "strategy": self.strategy.value,
            "total_pulls": total_pulls,
            "total_executions": len(self.history),
            "arms": {
                name: arm.to_dict()
                for name, arm in self.arms.items()
            },
            "best_arm": max(self.arms.values(), key=lambda a: a.mean_reward).name if self.arms else None
        }

    @calibrated_method("farfan_core.optimization.rl_strategy.RLStrategyOptimizer.save")
    def save(self, output_path: Path) -> None:
        """
        Save optimizer state to file.

        Args:
            output_path: Path to output file
        """
        state = {
            "optimizer_id": self.optimizer_id,
            "strategy": self.strategy.value,
            "created_at": self.created_at.isoformat(),
            "arms": {
                name: arm.to_dict()
                for name, arm in self.arms.items()
            },
            "statistics": self.get_statistics()
        }
```

```
2345:
2346:         output_path.parent.mkdir(parents=True, exist_ok=True)
2347:
2348:         with open(output_path, 'w', encoding='utf-8') as f:
2349:             json.dump(state, f, indent=2)
2350:
2351:         logger.info(f"Saved optimizer state to {output_path}")
2352:
2353:     @calibrated_method("farfan_core.optimization.rl_strategy.RLStrategyOptimizer.load")
2354:     def load(self, input_path: Path) -> None:
2355:         """
2356:         Load optimizer state from file.
2357:
2358:         Args:
2359:             input_path: Path to input file
2360:         """
2361:         with open(input_path, encoding='utf-8') as f:
2362:             state = json.load(f)
2363:
2364:         # Restore arms
2365:         self.arms.clear()
2366:         for name, arm_data in state["arms"].items():
2367:             arm = BanditArm(
2368:                 arm_id=arm_data["arm_id"],
2369:                 name=arm_data["name"],
2370:                 alpha=arm_data["alpha"],
2371:                 beta=arm_data["beta"],
2372:                 pulls=arm_data["pulls"],
2373:                 total_reward=arm_data["mean_reward"] * arm_data["pulls"],
2374:                 successes=int(arm_data["success_rate"] * arm_data["pulls"]),
2375:                 failures=arm_data["pulls"] - int(arm_data["success_rate"] * arm_data["pulls"]),
2376:                 total_duration_ms=arm_data["mean_duration_ms"] * arm_data["pulls"],
2377:                 total_tokens=arm_data["total_tokens"],
2378:                 total_cost_usd=arm_data["mean_cost_usd"] * arm_data["pulls"]
2379:             )
2380:             self.arms[name] = arm
2381:
2382:         logger.info(f"Loaded optimizer state from {input_path}")
2383:
2384:     @calibrated_method("farfan_core.optimization.rl_strategy.RLStrategyOptimizer.print_summary")
2385:     def print_summary(self) -> None:
2386:         """Print summary of optimizer state."""
2387:         stats = self.get_statistics()
2388:
2389:         print("\n" + "=" * 80)
2390:         print("ð\237¤\226 RL STRATEGY OPTIMIZER SUMMARY")
2391:         print("=" * 80)
2392:         print(f"Strategy: {self.strategy.value}")
2393:         print(f"Total Pulls: {stats['total_pulls']}")
2394:         print(f"Total Executions: {stats['total_executions']}")
2395:         print(f"Best Arm: {stats['best_arm']}")
2396:
2397:         print("\n" + "-" * 80)
2398:         print("Arm Performance:")
2399:         print("-" * 80)
2400:
```

```
        # Sort arms by mean reward
        sorted_arms = sorted(
            self.arms.values(),
            key=lambda a: a.mean_reward,
            reverse=True
        )

        for arm in sorted_arms:
            print(f"\n{arm.name}:")
            print(f"  Pulls: {arm.pulls}")
            print(f"  Mean Reward: {arm.mean_reward:.4f}")
            print(f"  Success Rate: {arm.success_rate:.2%}")
            print(f"  Mean Duration: {arm.mean_duration_ms:.2f}ms")
            print(f"  Mean Cost: ${arm.mean_cost_usd:.4f}")

        print("\n" + "=" * 80)


# â\234\205 AUDIT_VERIFIED: RL-Based Strategy Optimization Complete
# - Multi-armed bandit algorithms (Thompson Sampling, UCB1, Epsilon-Greedy)
# - Bayesian posterior tracking for continuous learning
# - Comprehensive performance metrics
# - Persistence for long-term optimization
# - Statistical analysis and reporting


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

"""
FARFAN Mechanistic Policy Pipeline - Patterns Module
=====================================================

Design patterns for robust distributed systems including:
- Saga pattern for compensating transactions
- Event tracking for observability

Author: FARFAN Team
Date: 2025-11-13
Version: 1.0.0
"""

from farfan_pipeline.patterns.event_tracking import (
    Event,
    EventCategory,
    EventLevel,
    EventSpan,
    EventTracker,
    get_global_tracker,
    record_event,
    span,
)
from farfan_pipeline.patterns.saga import (
    SagaEvent,
```

```
    SagaOrchestrator,
    SagaStatus,
    SagaStep,
    SagaStepStatus,
    compensate_api_call,
    compensate_database_insert,
    compensate_file_write,
)

__all__ = [
    # Event Tracking
    "Event",
    "EventCategory",
    "EventLevel",
    "EventSpan",
    "EventTracker",
    "get_global_tracker",
    "record_event",
    "span",
    # Saga Pattern
    "SagaEvent",
    "SagaOrchestrator",
    "SagaStatus",
    "SagaStep",
    "SagaStepStatus",
    "compensate_api_call",
    "compensate_database_insert",
    "compensate_file_write",
]


================================================================================
FILE: src/farfan_pipeline/patterns/event_tracking.py
================================================================================

"""
FARFAN Mechanistic Policy Pipeline - Event Tracking System
============================================================

Provides explicit event tracking with timestamps for debugging and audit purposes.

Features:
- Hierarchical event structure (parent-child relationships)
- Rich metadata capture
- Performance metrics
- Event filtering and querying
- Export to various formats (JSON, CSV, logs)

â\234\205 AUDIT_VERIFIED: Explicit event tracking with timestamps for debugging

Author: FARFAN Team
Date: 2025-11-13
Version: 1.0.0
"""

```

```
import csv
import json
import logging
from dataclasses import dataclass, field
from datetime import datetime
from enum import Enum
from pathlib import Path
from typing import Any
from uuid import uuid4

logger = logging.getLogger(__name__)


class EventLevel(Enum):
    """Event severity level."""
    DEBUG = "DEBUG"
    INFO = "INFO"
    WARNING = "WARNING"
    ERROR = "ERROR"
    CRITICAL = "CRITICAL"


class EventCategory(Enum):
    """Event category for classification."""
    SYSTEM = "system"
    EXECUTOR = "executor"
    PIPELINE = "pipeline"
    ANALYSIS = "analysis"
    PROCESSING = "processing"
    VALIDATION = "validation"
    AUDIT = "audit"
    PERFORMANCE = "performance"
    ERROR = "error"


@dataclass
class Event:
    """
    Represents a single trackable event in the pipeline.

    â\234\205 AUDIT_VERIFIED: Event with timestamp and full metadata capture
    """
    event_id: str = field(default_factory=lambda: str(uuid4()))
    timestamp: datetime = field(default_factory=datetime.utcnow)
    category: EventCategory = EventCategory.SYSTEM
    level: EventLevel = EventLevel.INFO
    source: str = ""
    message: str = ""
    metadata: dict[str, Any] = field(default_factory=dict)
    parent_event_id: str | None = None
    duration_ms: float | None = None
    tags: list[str] = field(default_factory=list)

    def to_dict(self) -> dict[str, Any]:
        """Convert event to dictionary."""
        return {
```

```
            "event_id": self.event_id,
            "timestamp": self.timestamp.isoformat(),
            "category": self.category.value,
            "level": self.level.value,
            "source": self.source,
            "message": self.message,
            "metadata": self.metadata,
            "parent_event_id": self.parent_event_id,
            "duration_ms": self.duration_ms,
            "tags": self.tags
        }

    def __str__(self) -> str:
        """String representation for logging."""
        timestamp_str = self.timestamp.strftime("%Y-%m-%d %H:%M:%S.%f")[:-3]
        duration_str = f" ({self.duration_ms:.2f}ms)" if self.duration_ms else ""
        return f"[{timestamp_str}] {self.level.value} [{self.category.value}] {self.source}: {self.message}{duration_str}"


@dataclass
class EventSpan:
    """
    Represents a time span for measuring operation duration.

    â\234\205 AUDIT_VERIFIED: Performance tracking with start/end timestamps
    """
    span_id: str = field(default_factory=lambda: str(uuid4()))
    name: str = ""
    category: EventCategory = EventCategory.PERFORMANCE
    parent_span_id: str | None = None
    start_time: datetime = field(default_factory=datetime.utcnow)
    end_time: datetime | None = None
    metadata: dict[str, Any] = field(default_factory=dict)
    tags: list[str] = field(default_factory=list)

    @property
    def duration_ms(self) -> float | None:
        """Calculate duration in milliseconds."""
        if self.end_time:
            delta = self.end_time - self.start_time
            return delta.total_seconds() * 1000
        return None

    @property
    def is_complete(self) -> bool:
        """Check if span is complete."""
        return self.end_time is not None

    def complete(self, metadata: dict[str, Any] | None = None) -> None:
        """Mark span as complete."""
        self.end_time = datetime.utcnow()
        if metadata:
            self.metadata.update(metadata)

    def to_event(self) -> Event:
        """Convert span to event."""
```

```
2625:         return Event(
2626:             event_id=self.span_id,
2627:             timestamp=self.start_time,
2628:             category=self.category,
2629:             level=EventLevel.INFO,
2630:             source=f"span:{self.name}",
2631:             message=f"Completed: {self.name}",
2632:             metadata=self.metadata,
2633:             parent_event_id=self.parent_span_id,
2634:             duration_ms=self.duration_ms,
2635:             tags=self.tags
2636:         )
2637:
2638:
2639: class EventTracker:
2640:     """
2641:     Central event tracking system for the FARFAN pipeline.
2642:
2643:     â\234\205 AUDIT_VERIFIED: Explicit event tracking with timestamps for debugging
2644:
2645:     Features:
2646:     - Event recording with automatic timestamps
2647:     - Hierarchical event organization
2648:     - Performance span tracking
2649:     - Event filtering and querying
2650:     - Export to multiple formats
2651:
2652:     Usage:
2653:         >>> tracker = EventTracker()
2654:         >>> tracker.record_event(
2655:         ...     category=EventCategory.EXECUTOR,
2656:         ...     source="D1Q1_Executor",
2657:         ...     message="Started execution"
2658:         ... )
2659:         >>> with tracker.span("process_policy"):
2660:         ...     # Do work
2661:         ...     pass
2662:     """
2663:
2664:     def __init__(self, name: str = "FARFAN Pipeline") -> None:
2665:         """
2666:         Initialize event tracker.
2667:
2668:         Args:
2669:             name: Name of the tracking session
2670:         """
2671:         self.name = name
2672:         self.events: list[Event] = []
2673:         self.spans: dict[str, EventSpan] = {}
2674:         self.session_id = str(uuid4())
2675:         self.started_at = datetime.utcnow()
2676:
2677:     def record_event(
2678:         self,
2679:         category: EventCategory,
2680:         source: str,
```

```
        message: str,
        level: EventLevel = EventLevel.INFO,
        metadata: dict[str, Any] | None = None,
        parent_event_id: str | None = None,
        tags: list[str] | None = None
    ) -> Event:
        """
        Record an event.

        Args:
            category: Event category
            source: Source of the event (e.g., executor name, module name)
            message: Event message
            level: Event severity level
            metadata: Additional metadata
            parent_event_id: Optional parent event ID for hierarchical tracking
            tags: Optional tags for filtering

        Returns:
            Created event
        """
        event = Event(
            category=category,
            level=level,
            source=source,
            message=message,
            metadata=metadata or {},
            parent_event_id=parent_event_id,
            tags=tags or []
        )

        self.events.append(event)

        # Log to standard logger
        log_fn = {
            EventLevel.DEBUG: logger.debug,
            EventLevel.INFO: logger.info,
            EventLevel.WARNING: logger.warning,
            EventLevel.ERROR: logger.error,
            EventLevel.CRITICAL: logger.critical
        }[level]

        log_fn(str(event))

        return event

    def start_span(
        self,
        name: str,
        category: EventCategory = EventCategory.PERFORMANCE,
        parent_span_id: str | None = None,
        metadata: dict[str, Any] | None = None,
        tags: list[str] | None = None
    ) -> EventSpan:
        """
        Start a new performance span.
```

```
2737:
2738:         Args:
2739:             name: Span name
2740:             category: Event category
2741:             parent_span_id: Optional parent span ID
2742:             metadata: Additional metadata
2743:             tags: Optional tags
2744:
2745:         Returns:
2746:             Started span
2747:         """
2748:         span = EventSpan(
2749:             name=name,
2750:             category=category,
2751:             parent_span_id=parent_span_id,
2752:             metadata=metadata or {},
2753:             tags=tags or []
2754:         )
2755:
2756:         self.spans[span.span_id] = span
2757:
2758:         self.record_event(
2759:             category=category,
2760:             source=f"span:{name}",
2761:             message=f"Started: {name}",
2762:             level=EventLevel.DEBUG,
2763:             parent_event_id=parent_span_id,
2764:             tags=tags
2765:         )
2766:
2767:         return span
2768:
2769:     def complete_span(
2770:         self,
2771:         span: EventSpan,
2772:         metadata: dict[str, Any] | None = None
2773:     ) -> Event:
2774:         """
2775:         Complete a span and record its event.
2776:
2777:         Args:
2778:             span: Span to complete
2779:             metadata: Additional metadata
2780:
2781:         Returns:
2782:             Event created from span
2783:         """
2784:         span.complete(metadata)
2785:         event = span.to_event()
2786:         self.events.append(event)
2787:
2788:         logger.debug(str(event))
2789:
2790:         return event
2791:
2792:     def span(
```

```
2793:         self,
2794:         name: str,
2795:         category: EventCategory = EventCategory.PERFORMANCE,
2796:         parent_span_id: str | None = None,
2797:         metadata: dict[str, Any] | None = None,
2798:         tags: list[str] | None = None
2799:     ):
2800:         """
2801:         Context manager for automatic span tracking.
2802:
2803:         Args:
2804:             name: Span name
2805:             category: Event category
2806:             parent_span_id: Optional parent span ID
2807:             metadata: Additional metadata
2808:             tags: Optional tags
2809:
2810:         Yields:
2811:             EventSpan object
2812:
2813:         Usage:
2814:             >>> with tracker.span("process_policy") as span:
2815:             ...     # Do work
2816:             ...     span.metadata["records_processed"] = 100
2817:         """
2818:         span = self.start_span(name, category, parent_span_id, metadata, tags)
2819:
2820:         try:
2821:             yield span
2822:         except Exception as e:
2823:             span.metadata["error"] = str(e)
2824:             span.metadata["error_type"] = type(e).__name__
2825:             self.record_event(
2826:                 category=EventCategory.ERROR,
2827:                 source=f"span:{name}",
2828:                 message=f"Failed: {name} - {e}",
2829:                 level=EventLevel.ERROR,
2830:                 parent_event_id=span.span_id
2831:             )
2832:             raise
2833:         finally:
2834:             self.complete_span(span)
2835:
2836:     def filter_events(
2837:         self,
2838:         category: EventCategory | None = None,
2839:         level: EventLevel | None = None,
2840:         source: str | None = None,
2841:         start_time: datetime | None = None,
2842:         end_time: datetime | None = None,
2843:         tags: list[str] | None = None
2844:     ) -> list[Event]:
2845:         """
2846:         Filter events by criteria.
2847:
2848:         Args:
```

```
2849:             category: Filter by category
2850:             level: Filter by level
2851:             source: Filter by source (exact match or contains)
2852:             start_time: Filter events after this time
2853:             end_time: Filter events before this time
2854:             tags: Filter by tags (any match)
2855:
2856:         Returns:
2857:             Filtered list of events
2858:         """
2859:         filtered = self.events
2860:
2861:         if category:
2862:             filtered = [e for e in filtered if e.category == category]
2863:
2864:         if level:
2865:             filtered = [e for e in filtered if e.level == level]
2866:
2867:         if source:
2868:             filtered = [e for e in filtered if source in e.source]
2869:
2870:         if start_time:
2871:             filtered = [e for e in filtered if e.timestamp >= start_time]
2872:
2873:         if end_time:
2874:             filtered = [e for e in filtered if e.timestamp <= end_time]
2875:
2876:         if tags:
2877:             filtered = [e for e in filtered if any(tag in e.tags for tag in tags)]
2878:
2879:         return filtered
2880:
2881:     def get_statistics(self) -> dict[str, Any]:
2882:         """
2883:         Get statistics about recorded events.
2884:
2885:         Returns:
2886:             Dictionary with statistics
2887:         """
2888:         if not self.events:
2889:             return {
2890:                 "total_events": 0,
2891:                 "session_duration_s": (datetime.utcnow() - self.started_at).total_seconds()
2892:             }
2893:
2894:         return {
2895:             "session_id": self.session_id,
2896:             "session_name": self.name,
2897:             "session_duration_s": (datetime.utcnow() - self.started_at).total_seconds(),
2898:             "total_events": len(self.events),
2899:             "total_spans": len(self.spans),
2900:             "by_category": {
2901:                 cat.value: len([e for e in self.events if e.category == cat])
2902:                 for cat in EventCategory
2903:             },
2904:             "by_level": {
```

```
                level.value: len([e for e in self.events if e.level == level])
                for level in EventLevel
            },
            "performance_spans": [
                {
                    "name": span.name,
                    "duration_ms": span.duration_ms,
                    "complete": span.is_complete
                }
                for span in self.spans.values()
                if span.is_complete
            ],
            "errors": len([e for e in self.events if e.level in [EventLevel.ERROR, EventLevel.CRITICAL]])
        }

    def export_json(self, output_path: Path) -> None:
        """
        Export events to JSON file.

        Args:
            output_path: Path to output file
        """
        data = {
            "session_id": self.session_id,
            "session_name": self.name,
            "started_at": self.started_at.isoformat(),
            "statistics": self.get_statistics(),
            "events": [e.to_dict() for e in self.events]
        }

        output_path.parent.mkdir(parents=True, exist_ok=True)

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

        logger.info(f"Exported {len(self.events)} events to {output_path}")

    def export_csv(self, output_path: Path) -> None:
        """
        Export events to CSV file.

        Args:
            output_path: Path to output file
        """
        output_path.parent.mkdir(parents=True, exist_ok=True)

        with open(output_path, 'w', newline='', encoding='utf-8') as f:
            if not self.events:
                return

            fieldnames = [
                "event_id", "timestamp", "category", "level",
                "source", "message", "duration_ms", "parent_event_id", "tags"
            ]

            writer = csv.DictWriter(f, fieldnames=fieldnames)
```

```
            writer.writeheader()

            for event in self.events:
                writer.writerow({
                    "event_id": event.event_id,
                    "timestamp": event.timestamp.isoformat(),
                    "category": event.category.value,
                    "level": event.level.value,
                    "source": event.source,
                    "message": event.message,
                    "duration_ms": event.duration_ms or "",
                    "parent_event_id": event.parent_event_id or "",
                    "tags": ",".join(event.tags)
                })

        logger.info(f"Exported {len(self.events)} events to {output_path}")

    def print_summary(self) -> None:
        """Print summary of tracked events."""
        stats = self.get_statistics()

        print("\n" + "=" * 80)
        print(f"ð\237\223\212 EVENT TRACKING SUMMARY: {self.name}")
        print("=" * 80)
        print(f"Session ID: {self.session_id}")
        print(f"Duration: {stats['session_duration_s']:.2f}s")
        print(f"Total Events: {stats['total_events']}")
        print(f"Total Spans: {stats['total_spans']}")
        print(f"Errors: {stats['errors']}")

        print("\n" + "-" * 80)
        print("Events by Category:")
        print("-" * 80)
        for category, count in stats['by_category'].items():
            if count > 0:
                print(f"  {category}: {count}")

        print("\n" + "-" * 80)
        print("Events by Level:")
        print("-" * 80)
        for level, count in stats['by_level'].items():
            if count > 0:
                print(f"  {level}: {count}")

        if stats['performance_spans']:
            print("\n" + "-" * 80)
            print("Performance Spans (Top 10 by duration):")
            print("-" * 80)
            sorted_spans = sorted(
                stats['performance_spans'],
                key=lambda x: x['duration_ms'] or 0,
                reverse=True
            )[:10]

            for span in sorted_spans:
                if span['duration_ms']:
```

```
                    print(f"  {span['name']}: {span['duration_ms']:.2f}ms")

        print("\n" + "=" * 80)


# Global tracker instance for convenience
_global_tracker: EventTracker | None = None


def get_global_tracker() -> EventTracker:
    """Get or create global event tracker."""
    global _global_tracker
    if _global_tracker is None:
        _global_tracker = EventTracker("FARFAN Global Tracker")
    return _global_tracker


def record_event(*args, **kwargs) -> Event:
    """Convenience function to record event on global tracker."""
    return get_global_tracker().record_event(*args, **kwargs)


def span(*args, **kwargs):
    """Convenience function to create span on global tracker."""
    return get_global_tracker().span(*args, **kwargs)


# â\234\205 AUDIT_VERIFIED: Event Tracking System Complete
# - Explicit timestamps on all events
# - Hierarchical event organization
# - Performance span tracking
# - Rich metadata capture
# - Multiple export formats
# - Global tracker for convenience


================================================================================
FILE: src/farfan_pipeline/patterns/saga.py
================================================================================

"""
FARFAN Mechanistic Policy Pipeline - Saga Pattern
=================================================

Implements the Saga pattern for managing distributed transactions and
compensating actions in critical pipeline operations.

The Saga pattern ensures data consistency across multiple operations by:
1. Breaking complex transactions into smaller steps
2. Providing compensating transactions for rollback
3. Maintaining audit trail of all actions
4. Supporting both forward and backward recovery

Reference: Garcia-Molina & Salem (1987) "Sagas"

```

```
Author: FARFAN Team
Date: 2025-11-13
Version: 1.0.0
"""

import logging
from collections.abc import Callable
from dataclasses import dataclass, field
from datetime import datetime
from enum import Enum
from typing import Any
from uuid import uuid4

logger = logging.getLogger(__name__)


class SagaStepStatus(Enum):
    """Status of a saga step."""
    PENDING = "pending"
    EXECUTING = "executing"
    COMPLETED = "completed"
    FAILED = "failed"
    COMPENSATING = "compensating"
    COMPENSATED = "compensated"
    COMPENSATION_FAILED = "compensation_failed"


class SagaStatus(Enum):
    """Overall saga status."""
    INITIALIZED = "initialized"
    IN_PROGRESS = "in_progress"
    COMPLETED = "completed"
    FAILED = "failed"
    COMPENSATING = "compensating"
    COMPENSATED = "compensated"
    COMPENSATION_FAILED = "compensation_failed"


@dataclass
class SagaStep:
    """
    Represents a single step in a saga with its compensating action.

    â\234\205 AUDIT_VERIFIED: Saga step with full compensation support
    """
    step_id: str
    name: str
    execute_fn: Callable[..., Any]
    compensate_fn: Callable[..., Any]
    status: SagaStepStatus = SagaStepStatus.PENDING
    result: Any | None = None
    error: Exception | None = None
    started_at: datetime | None = None
    completed_at: datetime | None = None
    compensated_at: datetime | None = None

```

```
    def execute(self, *args, **kwargs) -> Any:
        """
        Execute the step.

        Returns:
            Result of the step execution

        Raises:
            Exception: If execution fails
        """
        self.status = SagaStepStatus.EXECUTING
        self.started_at = datetime.utcnow()

        try:
            logger.info(f"Executing saga step: {self.name} (ID: {self.step_id})")
            self.result = self.execute_fn(*args, **kwargs)
            self.status = SagaStepStatus.COMPLETED
            self.completed_at = datetime.utcnow()
            logger.info(f"Completed saga step: {self.name}")
            return self.result
        except Exception as e:
            self.status = SagaStepStatus.FAILED
            self.error = e
            self.completed_at = datetime.utcnow()
            logger.error(f"Failed saga step: {self.name} - {e}")
            raise

    def compensate(self, *args, **kwargs) -> None:
        """
        Execute compensating action for this step.

        Raises:
            Exception: If compensation fails
        """
        if self.status != SagaStepStatus.COMPLETED:
            logger.warning(f"Cannot compensate step {self.name} with status {self.status}")
            return

        self.status = SagaStepStatus.COMPENSATING

        try:
            logger.info(f"Compensating saga step: {self.name} (ID: {self.step_id})")
            self.compensate_fn(self.result, *args, **kwargs)
            self.status = SagaStepStatus.COMPENSATED
            self.compensated_at = datetime.utcnow()
            logger.info(f"Compensated saga step: {self.name}")
        except Exception as e:
            self.status = SagaStepStatus.COMPENSATION_FAILED
            self.error = e
            logger.error(f"Failed to compensate saga step: {self.name} - {e}")
            raise


@dataclass
class SagaEvent:
    """
```

```
    Represents an event in the saga lifecycle.

    â\234\205 AUDIT_VERIFIED: Event tracking with timestamps for debugging
    """
    event_id: str = field(default_factory=lambda: str(uuid4()))
    saga_id: str = ""
    event_type: str = ""
    step_id: str | None = None
    step_name: str | None = None
    timestamp: datetime = field(default_factory=datetime.utcnow)
    data: dict[str, Any] = field(default_factory=dict)

    def to_dict(self) -> dict[str, Any]:
        """Convert event to dictionary."""
        return {
            "event_id": self.event_id,
            "saga_id": self.saga_id,
            "event_type": self.event_type,
            "step_id": self.step_id,
            "step_name": self.step_name,
            "timestamp": self.timestamp.isoformat(),
            "data": self.data
        }


class SagaOrchestrator:
    """
    Orchestrates saga execution with automatic compensation on failure.

    â\234\205 AUDIT_VERIFIED: Saga pattern for compensating actions in critical operations
    â\234\205 AUDIT_VERIFIED: Explicit event tracking with timestamps for debugging

    Usage:
        >>> saga = SagaOrchestrator(saga_id="process_policy_001")
        >>> saga.add_step("load_data", load_fn, cleanup_fn)
        >>> saga.add_step("process", process_fn, rollback_fn)
        >>> result = saga.execute()
    """

    def __init__(self, saga_id: str | None = None, name: str = "Unnamed Saga") -> None:
        """
        Initialize saga orchestrator.

        Args:
            saga_id: Unique identifier for the saga (auto-generated if None)
            name: Human-readable name for the saga
        """
        self.saga_id = saga_id or str(uuid4())
        self.name = name
        self.steps: list[SagaStep] = []
        self.status = SagaStatus.INITIALIZED
        self.events: list[SagaEvent] = []
        self.created_at = datetime.utcnow()
        self.completed_at: datetime | None = None

    def add_step(
```

```
        self,
        name: str,
        execute_fn: Callable[..., Any],
        compensate_fn: Callable[..., Any],
        step_id: str | None = None
    ) -> "SagaOrchestrator":
        """
        Add a step to the saga.

        Args:
            name: Step name
            execute_fn: Function to execute the step
            compensate_fn: Function to compensate the step
            step_id: Optional step ID (auto-generated if None)

        Returns:
            Self for chaining
        """
        step = SagaStep(
            step_id=step_id or str(uuid4()),
            name=name,
            execute_fn=execute_fn,
            compensate_fn=compensate_fn
        )
        self.steps.append(step)
        self._record_event("step_added", step.step_id, step.name, {"step_count": len(self.steps)})
        return self

    def execute(self, *args, **kwargs) -> dict[str, Any]:
        """
        Execute all saga steps in sequence with automatic compensation on failure.

        Args:
            *args: Arguments to pass to step execution functions
            **kwargs: Keyword arguments to pass to step execution functions

        Returns:
            Dictionary with execution results

        Raises:
            Exception: If saga execution fails and compensation also fails
        """
        self.status = SagaStatus.IN_PROGRESS
        self._record_event("saga_started", data={"step_count": len(self.steps)})

        executed_steps: list[SagaStep] = []

        try:
            # Execute all steps in order
            for step in self.steps:
                logger.info(f"[Saga: {self.name}] Executing step: {step.name}")
                self._record_event("step_started", step.step_id, step.name)

                result = step.execute(*args, **kwargs)
                executed_steps.append(step)

```

```
                self._record_event(
                    "step_completed",
                    step.step_id,
                    step.name,
                    {"result_type": type(result).__name__}
                )

            # All steps succeeded
            self.status = SagaStatus.COMPLETED
            self.completed_at = datetime.utcnow()
            self._record_event("saga_completed", data={"duration_s": self._duration()})

            logger.info(f"[Saga: {self.name}] Completed successfully")

            return {
                "saga_id": self.saga_id,
                "status": self.status.value,
                "steps_completed": len(executed_steps),
                "results": [step.result for step in executed_steps]
            }

        except Exception as e:
            # A step failed - trigger compensation
            self.status = SagaStatus.FAILED
            logger.error(f"[Saga: {self.name}] Failed: {e}")
            self._record_event("saga_failed", data={"error": str(e)})

            # Compensate in reverse order
            return self._compensate(executed_steps, original_error=e)

    def _compensate(
        self,
        executed_steps: list[SagaStep],
        original_error: Exception
    ) -> dict[str, Any]:
        """
        Execute compensating actions for all completed steps.

        Args:
            executed_steps: Steps that were successfully executed
            original_error: The error that triggered compensation

        Returns:
            Dictionary with compensation results

        Raises:
            Exception: If compensation fails
        """
        self.status = SagaStatus.COMPENSATING
        self._record_event("compensation_started", data={"steps_to_compensate": len(executed_steps)})

        logger.warning(f"[Saga: {self.name}] Compensating {len(executed_steps)} steps")

        compensation_errors = []

        # Compensate in reverse order
```

```
        for step in reversed(executed_steps):
            try:
                self._record_event("compensation_step_started", step.step_id, step.name)
                step.compensate()
                self._record_event("compensation_step_completed", step.step_id, step.name)
            except Exception as comp_error:
                compensation_errors.append({
                    "step": step.name,
                    "error": str(comp_error)
                })
                logger.error(f"[Saga: {self.name}] Compensation failed for step {step.name}: {comp_error}")
                self._record_event(
                    "compensation_step_failed",
                    step.step_id,
                    step.name,
                    {"error": str(comp_error)}
                )

        if compensation_errors:
            self.status = SagaStatus.COMPENSATION_FAILED
            self._record_event("compensation_failed", data={"errors": compensation_errors})
            raise Exception(
                f"Saga compensation failed. Original error: {original_error}. "
                f"Compensation errors: {compensation_errors}"
            )
        else:
            self.status = SagaStatus.COMPENSATED
            self.completed_at = datetime.utcnow()
            self._record_event("compensation_completed", data={"duration_s": self._duration()})
            logger.info(f"[Saga: {self.name}] Compensation completed successfully")

            return {
                "saga_id": self.saga_id,
                "status": self.status.value,
                "original_error": str(original_error),
                "steps_compensated": len(executed_steps),
                "compensation_errors": compensation_errors
            }

    def _record_event(
        self,
        event_type: str,
        step_id: str | None = None,
        step_name: str | None = None,
        data: dict[str, Any] | None = None
    ) -> None:
        """Record an event in the saga lifecycle."""
        event = SagaEvent(
            saga_id=self.saga_id,
            event_type=event_type,
            step_id=step_id,
            step_name=step_name,
            data=data or {}
        )
        self.events.append(event)

```

```
    def _duration(self) -> float:
        """Calculate saga duration in seconds."""
        if self.completed_at:
            return (self.completed_at - self.created_at).total_seconds()
        return (datetime.utcnow() - self.created_at).total_seconds()

    def get_audit_trail(self) -> list[dict[str, Any]]:
        """
        Get complete audit trail of saga execution.

        Returns:
            List of events in chronological order
        """
        return [event.to_dict() for event in self.events]

    def to_dict(self) -> dict[str, Any]:
        """Convert saga to dictionary for serialization."""
        return {
            "saga_id": self.saga_id,
            "name": self.name,
            "status": self.status.value,
            "created_at": self.created_at.isoformat(),
            "completed_at": self.completed_at.isoformat() if self.completed_at else None,
            "duration_s": self._duration(),
            "steps": [
                {
                    "step_id": step.step_id,
                    "name": step.name,
                    "status": step.status.value,
                    "started_at": step.started_at.isoformat() if step.started_at else None,
                    "completed_at": step.completed_at.isoformat() if step.completed_at else None,
                    "compensated_at": step.compensated_at.isoformat() if step.compensated_at else None,
                    "error": str(step.error) if step.error else None
                }
                for step in self.steps
            ],
            "events": self.get_audit_trail()
        }


# Example compensating functions for common operations
def compensate_file_write(file_path: str, original_content: str | None = None) -> None:
    """Compensate a file write operation by restoring original content or deleting."""
    import os
    if original_content is not None:
        with open(file_path, 'w') as f:
            f.write(original_content)
    elif os.path.exists(file_path):
        os.remove(file_path)


def compensate_database_insert(db_connection, table: str, record_id: Any) -> None:
    """Compensate a database insert by deleting the record.

    Note: This is a simplified example. In production, validate table name
    against a whitelist to prevent SQL injection attacks.
```

```
    """
    # Validate table name against allowed tables
    # In a real implementation, this should be configured per application
    allowed_tables = {"users", "orders", "transactions", "policies", "executors"}
    if table not in allowed_tables:
        raise ValueError(f"Invalid table name: {table}. Allowed tables: {allowed_tables}")

    cursor = db_connection.cursor()
    cursor.execute(f"DELETE FROM {table} WHERE id = ?", (record_id,))
    db_connection.commit()


def compensate_api_call(api_client, endpoint: str, created_id: str) -> None:
    """Compensate an API create call with a delete call."""
    api_client.delete(f"{endpoint}/{created_id}")


# â\234\205 AUDIT_VERIFIED: Saga Pattern Implementation Complete
# - Supports forward execution with compensation on failure
# - Maintains complete audit trail with timestamps
# - Handles compensation failures gracefully
# - Provides serialization for persistence and debugging


================================================================================
FILE: src/farfan_pipeline/processing/Copia de __init__.py
================================================================================

"""
SPC (Smart Policy Chunks) Ingestion - Canonical Phase-One
============================================================

This module provides the canonical phase-one ingestion pipeline for processing
development plans into smart policy chunks with comprehensive analysis.

Main exports:
- CPPIngestionPipeline: Primary ingestion pipeline (for compatibility)
- StrategicChunkingSystem: Core chunking system from smart_policy_chunks_canonic_phase_one

The pipeline performs:
1. Document preprocessing and structural analysis
2. Topic modeling and knowledge graph construction
3. Causal chain extraction
4. Temporal, argumentative, and discourse analysis
5. Smart chunk creation with inter-chunk relationships
6. Quality validation and strategic ranking
"""

import importlib.util
import logging
import unicodedata  # For NFC normalization
from pathlib import Path
from typing import Any

from farfan_pipeline.config.paths import QUESTIONNAIRE_FILE
```

```
from farfan_pipeline.processing.cpp_ingestion.models import CanonPolicyPackage
from farfan_pipeline.processing.spc_ingestion.converter import SmartChunkConverter
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.processing.spc_ingestion.quality_gates import SPCQualityGates

logger = logging.getLogger(__name__)

# Load smart_policy_chunks_canonic_phase_one without sys.path manipulation
_root = Path(__file__).parent.parent.parent.parent.parent
_module_path = _root / "scripts" / "smart_policy_chunks_canonic_phase_one.py"

spec = importlib.util.spec_from_file_location(
    "smart_policy_chunks_canonic_phase_one",
    _module_path
)
if spec and spec.loader:
    _module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(_module)
    StrategicChunkingSystem = _module.StrategicChunkingSystem
else:
    raise ImportError(f"Cannot load smart_policy_chunks_canonic_phase_one from {_module_path}")


class CPPIngestionPipeline:
    """
    SPC ingestion pipeline with orchestrator-compatible output.

    This class provides the canonical phase-one ingestion pipeline:
    1. Processes documents through StrategicChunkingSystem (15-phase analysis)
    2. Converts SmartPolicyChunk output to CanonPolicyPackage format
    3. Returns orchestrator-ready CanonPolicyPackage

    The pipeline ensures 100% alignment between SPC phase-one output and
    what the orchestrator expects to receive.

    Questionnaire Input Contract (SIN_CARRETA compliance):
    ------------------------------------------------------
    - questionnaire_path is an EXPLICIT input (defaults to canonical path)
    - Must be deterministic, auditable, and manifest-tracked
    - No hidden filesystem dependencies
    """

    def __init__(
        self,
        questionnaire_path: Path | None = None,
        enable_runtime_validation: bool = True,
    ) -> None:
        """
        Initialize the SPC ingestion pipeline with converter.

        Args:
            questionnaire_path: Optional path to questionnaire file.
                                If None, uses canonical path from farfan_core.config.paths.QUESTIONNAIRE_FILE
            enable_runtime_validation: Enable WiringValidator for runtime contract checking
        """
        logger.info("Initializing CPPIngestionPipeline with StrategicChunkingSystem")
```

```
3577:
3578:         # Store questionnaire path for manifest traceability
3579:         if questionnaire_path is None:
3580:             questionnaire_path = QUESTIONNAIRE_FILE
3581:
3582:         self.questionnaire_path = questionnaire_path
3583:         logger.info(f"Questionnaire path: {self.questionnaire_path}")
3584:
3585:         self.chunking_system = StrategicChunkingSystem()
3586:         self.converter = SmartChunkConverter()
3587:         self.quality_gates = SPCQualityGates()
3588:
3589:         # Initialize WiringValidator for runtime contract validation
3590:         self.enable_runtime_validation = enable_runtime_validation
3591:         if enable_runtime_validation:
3592:             try:
3593:                 from farfan_pipeline.core.wiring.validation import WiringValidator
3594:                 self.wiring_validator = WiringValidator()
3595:                 logger.info("WiringValidator enabled for runtime contract checking")
3596:             except ImportError:
3597:                 logger.warning(
3598:                     "WiringValidator not available. Runtime validation disabled."
3599:                 )
3600:                 self.wiring_validator = None
3601:         else:
3602:             self.wiring_validator = None
3603:
3604:         logger.info("Pipeline initialized successfully")
3605:
3606:     def _load_document_text(self, document_path: Path) -> str:
3607:         """
3608:         Load document text from PDF, TXT, or MD files.
3609:
3610:         Args:
3611:             document_path: Path to document file
3612:
3613:         Returns:
3614:             Extracted text content
3615:
3616:         Raises:
3617:             ValueError: If file type is unsupported
3618:             IOError: If file cannot be read
3619:         """
3620:         suffix = document_path.suffix.lower()
3621:
3622:         if suffix == '.pdf':
3623:             # Use PyMuPDF (fitz) for PDF extraction
3624:             try:
3625:                 import fitz  # PyMuPDF
3626:                 doc = fitz.open(document_path)
3627:                 text_parts = []
3628:                 for page in doc:
3629:                     text_parts.append(page.get_text())
3630:                 doc.close()
3631:                 text = '\n'.join(text_parts)
3632:
```

```
3633:                 # Normalize to NFC for deterministic hashing and span calculation
3634:                 text = unicodedata.normalize('NFC', text)
3635:
3636:                 logger.info(f"Extracted {len(text)} characters from PDF ({len(text_parts)} pages)")
3637:                 return text
3638:             except ImportError:
3639:                 logger.error("PyMuPDF (fitz) not available for PDF extraction")
3640:                 raise OSError(
3641:                     "PDF extraction requires PyMuPDF (install with: pip install PyMuPDF). "
3642:                     "Alternatively, convert PDF to text manually."
3643:                 )
3644:             except Exception as e:
3645:                 logger.error(f"Failed to extract PDF: {e}")
3646:                 raise OSError(f"PDF extraction failed: {e}")
3647:
3648:         elif suffix in ['.txt', '.md']:
3649:             # Plain text or markdown
3650:             try:
3651:                 with open(document_path, encoding='utf-8') as f:
3652:                     text = f.read()
3653:
3654:                 # Normalize to NFC for deterministic hashing and span calculation
3655:                 text = unicodedata.normalize('NFC', text)
3656:
3657:                 logger.info(f"Loaded {len(text)} characters from {suffix} file")
3658:                 return text
3659:             except OSError as e:
3660:                 logger.error(f"Failed to read text file: {e}")
3661:                 raise
3662:
3663:         else:
3664:             raise ValueError(
3665:                 f"Unsupported file type: {suffix}. "
3666:                 f"Supported types: .pdf, .txt, .md"
3667:             )
3668:
3669:     async def process(
3670:         self,
3671:         document_path: Path,
3672:         document_id: str = None,
3673:         title: str = None
3674:     ) -> CanonPolicyPackage:
3675:         """
3676:         Process a document through the complete SPC pipeline.
3677:
3678:         Args:
3679:             document_path: Path to input document
3680:             document_id: Optional document identifier
3681:             title: Optional document title
3682:             max_chunks: Maximum number of chunks to generate
3683:
3684:         Returns:
3685:             CanonPolicyPackage: Orchestrator-ready policy package with:
3686:                 - ChunkGraph with all chunks and relationships
3687:                 - PolicyManifest with axes/programs/projects
3688:                 - QualityMetrics from SPC analysis
```

```
3689:                 - IntegrityIndex for verification
3690:                 - Rich SPC data preserved in metadata
3691:
3692:         Raises:
3693:             ValueError: If document is empty or invalid
3694:             IOError: If document cannot be read
3695:         """
3696:         logger.info(f"Processing document: {document_path}")
3697:
3698:         # Quality gate: Validate input file
3699:         validation_input = self.quality_gates.validate_input(document_path)
3700:         if not validation_input["passed"]:
3701:             raise ValueError(
3702:                 f"SPC input validation failed: {validation_input['failures']}"
3703:             )
3704:         logger.info(f"Input validation passed (file size: {validation_input['file_size_bytes']} bytes)")
3705:
3706:         # Load document text (supports PDF, TXT, MD)
3707:         try:
3708:             document_text = self._load_document_text(document_path)
3709:         except (OSError, ValueError) as e:
3710:             logger.error(f"Failed to load document: {e}")
3711:             raise
3712:
3713:         if not document_text or not document_text.strip():
3714:             raise ValueError(f"Document text is empty after extraction: {document_path}")
3715:
3716:         logger.info(f"Document loaded: {len(document_text)} characters")
3717:
3718:         # Prepare metadata
3719:         metadata = {
3720:             'document_id': document_id or str(document_path.stem),
3721:             'title': title or document_path.name,
3722:             'version': 'v3.0',
3723:             'source_path': str(document_path)
3724:         }
3725:
3726:         # Process through chunking system (15-phase analysis)
3727:         logger.info("Starting StrategicChunkingSystem.generate_smart_chunks()")
3728:         smart_chunks = self.chunking_system.generate_smart_chunks(document_text, metadata)
3729:         logger.info(f"Generated {len(smart_chunks)} SmartPolicyChunks")
3730:
3731:         # Quality gate: Validate chunks
3732:         chunk_dicts = [
3733:             {
3734:                 "text": c.text,
3735:                 "chunk_id": c.chunk_id,
3736:                 "strategic_importance": c.strategic_importance,
3737:                 "quality_score": c.confidence_metrics.get("overall_confidence", ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.__init__.CPPInge
stionPipeline._load_document_text", "auto_param_L243_80", 0.0)),
3738:             }
3739:             for c in smart_chunks
3740:         ]
3741:         validation_chunks = self.quality_gates.validate_chunks(chunk_dicts)
3742:         if not validation_chunks["passed"]:
3743:             raise ValueError(
```

```
3744:                 f"SPC chunk validation failed: {validation_chunks['failures']}"
3745:             )
3746:         if validation_chunks.get("warnings"):
3747:             logger.warning(f"Chunk validation warnings: {validation_chunks['warnings'][:3]}")
3748:         logger.info(f"Chunk validation passed ({validation_chunks['chunk_count']} chunks)")
3749:
3750:
3751:         # Convert to CanonPolicyPackage
3752:         logger.info("Converting SmartPolicyChunks to CanonPolicyPackage")
3753:         canon_package = self.converter.convert_to_canon_package(smart_chunks, metadata)
3754:
3755:         # Log quality metrics
3756:         if canon_package.quality_metrics:
3757:             logger.info(
3758:                 f"Quality metrics - "
3759:                 f"provenance: {canon_package.quality_metrics.provenance_completeness:.2%}, "
3760:                 f"coherence: {canon_package.quality_metrics.structural_consistency:.2%}, "
3761:                 f"coverage: {canon_package.quality_metrics.chunk_context_coverage:.2%}"
3762:             )
3763:
3764:         # RUNTIME VALIDATION: Validate CPP â\206\222 Adapter contract
3765:         if self.wiring_validator is not None:
3766:             logger.info("Validating CPP â\206\222 Adapter contract (runtime)")
3767:             try:
3768:                 # Convert CanonPolicyPackage to dict for validation
3769:                 cpp_dict = self._canon_package_to_dict(canon_package)
3770:                 self.wiring_validator.validate_cpp_to_adapter(cpp_dict)
3771:                 logger.info("â\234\223 CPP â\206\222 Adapter contract validation passed")
3772:             except Exception as e:
3773:                 logger.error(f"CPP â\206\222 Adapter contract validation failed: {e}")
3774:                 raise ValueError(
3775:                     f"Runtime contract violation at CPP â\206\222 Adapter boundary: {e}"
3776:                 ) from e
3777:
3778:         logger.info(f"Pipeline complete: {len(canon_package.chunk_graph.chunks)} chunks in package")
3779:         return canon_package
3780:
3781:     def _canon_package_to_dict(self, canon_package: CanonPolicyPackage) -> dict[str, Any]:
3782:         """Convert CanonPolicyPackage to dict for WiringValidator.
3783:
3784:         Args:
3785:             canon_package: CanonPolicyPackage to convert
3786:
3787:         Returns:
3788:             Dict representation for validation
3789:         """
3790:         # Extract chunks as list of dicts
3791:         chunks = []
3792:         if hasattr(canon_package, 'chunk_graph') and canon_package.chunk_graph:
3793:             for chunk_id, chunk in canon_package.chunk_graph.chunks.items():
3794:                 chunk_dict = {
3795:                     "chunk_id": chunk_id,
3796:                     "text": chunk.text if hasattr(chunk, 'text') else "",
3797:                     "text_span": {
3798:                         "start": chunk.text_span.start if hasattr(chunk, 'text_span') else 0,
3799:                         "end": chunk.text_span.end if hasattr(chunk, 'text_span') else 0,
```

```
                    } if hasattr(chunk, 'text_span') else {"start": 0, "end": 0},
                }
                chunks.append(chunk_dict)

        # Build validation dict
        return {
            "schema_version": canon_package.schema_version if hasattr(canon_package, 'schema_version') else "SPC-2025.1",
            "chunks": chunks,
            "chunk_count": len(chunks),
            "quality_metrics": {
                "provenance_completeness": (
                    canon_package.quality_metrics.provenance_completeness
                    if hasattr(canon_package, 'quality_metrics') and canon_package.quality_metrics
                    else ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.__init__.CPPIngestionPipeline._canon_package_to_dict", "auto_param_L319
_25", 0.0)
                ),
                "structural_consistency": (
                    canon_package.quality_metrics.structural_consistency
                    if hasattr(canon_package, 'quality_metrics') and canon_package.quality_metrics
                    else ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.__init__.CPPIngestionPipeline._canon_package_to_dict", "auto_param_L324
_25", 0.0)
                ),
            } if hasattr(canon_package, 'quality_metrics') else {},
        }


__all__ = [
    'CPPIngestionPipeline',
    'StrategicChunkingSystem',
    'SmartChunkConverter',
]



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

"""Processing modules for data transformation and analysis."""

```