```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 13
================================================================================
Generated: 2025-12-07T06:17:20.655468
Files in this batch: 17
================================================================================


================================================================================
FILE: src/farfan_pipeline/core/method_inventory.py
================================================================================

"""
Static Method Inventory Generator - Core AST Analysis (Stages A-D).

This module implements the foundational logic to scan the codebase, parse files,
and extract raw method nodes. It is the first step in the inventory generation pipeline.
"""
from __future__ import annotations

import ast
import sys
from dataclasses import dataclass
from pathlib import Path

SRC_ROOT = Path("farfan_core/farfan_core/core")
DEFAULT_OUTPUT = Path("farfan_core/farfan_core/artifacts/calibration/method_inventory.json")

from farfan_pipeline.core.method_inventory_types import (
    GovernanceFlags,
    SignatureDescriptor,
)

# ... (Stages A-D remain unchanged) ...

# --- Stage F: Signature ---

def classify_default_type(default_node: ast.expr) -> str:
    """Classify the type of default value."""
    if isinstance(default_node, ast.Constant):
        return "literal"
    elif isinstance(default_node, (ast.Name, ast.Attribute)):
        return "expression"
    else:
        return "complex"

def extract_parameter_info(
    arg: ast.arg,
    default: ast.expr | None,
    is_kwonly: bool = False
) -> ParameterDescriptor:
    """Extract detailed parameter information from AST."""
    from farfan_pipeline.core.method_inventory_types import ParameterDescriptor

    name = arg.arg

```

```
    type_hint = None
    if arg.annotation:
        try:
            type_hint = ast.unparse(arg.annotation)
        except AttributeError:
            type_hint = "Any"

    has_default = default is not None
    required = not has_default

    default_value = None
    default_type = None
    default_source = None

    if has_default:
        try:
            default_source = ast.unparse(default)
            default_value = default_source
            default_type = classify_default_type(default)
        except AttributeError:
            default_value = "<unparseable>"
            default_type = "complex"
            default_source = "<unparseable>"

    return ParameterDescriptor(
        name=name,
        type_hint=type_hint,
        has_default=has_default,
        required=required,
        default_value=default_value,
        default_type=default_type,
        default_source=default_source
    )

def should_parametrize_method(func_def: ast.FunctionDef | ast.AsyncFunctionDef) -> bool:
    """Determine if a method requires parameterization."""
    if func_def.name in ('__init__', '__repr__', '__str__', '__eq__'):
        return False

    all_args = func_def.args.args + func_def.args.kwonlyargs
    non_self_args = [a for a in all_args if a.arg not in ('self', 'cls')]

    return len(non_self_args) > 0

def extract_signature(func_def: ast.FunctionDef | ast.AsyncFunctionDef) -> SignatureDescriptor:
    from farfan_pipeline.core.method_inventory_types import SignatureDescriptor

    args = [a.arg for a in func_def.args.posonlyargs + func_def.args.args]
    kwargs = [a.arg for a in func_def.args.kwonlyargs]

    returns = "None"
    if func_def.returns:
        try:
            returns = ast.unparse(func_def.returns)
        except AttributeError:
            returns = "Complex"
```

```
113:
114:     accepts_executor_config = False
115:     all_args = func_def.args.posonlyargs + func_def.args.args + func_def.args.kwonlyargs
116:     for arg in all_args:
117:         if arg.annotation:
118:             try:
119:                 ann_str = ast.unparse(arg.annotation)
120:                 if "ExecutorConfig" in ann_str or "RuntimeConfig" in ann_str:
121:                     accepts_executor_config = True
122:             except AttributeError:
123:                 pass
124:
125:     requiere_parametrizacion = should_parametrize_method(func_def)
126:     input_parameters = None
127:
128:     if requiere_parametrizacion:
129:         input_parameters = []
130:
131:         posonly_args = func_def.args.posonlyargs
132:         posonly_defaults = []
133:         if func_def.args.defaults:
134:             num_posonly_with_defaults = max(0, len(func_def.args.defaults) - len(func_def.args.args))
135:             posonly_defaults = func_def.args.defaults[:num_posonly_with_defaults]
136:
137:         for i, arg in enumerate(posonly_args):
138:             if arg.arg in ('self', 'cls'):
139:                 continue
140:             default_idx = i - (len(posonly_args) - len(posonly_defaults))
141:             default = posonly_defaults[default_idx] if default_idx >= 0 else None
142:             input_parameters.append(extract_parameter_info(arg, default))
143:
144:         pos_args = func_def.args.args
145:         pos_defaults = func_def.args.defaults if func_def.args.defaults else []
146:         num_no_default = len(pos_args) - len(pos_defaults)
147:
148:         for i, arg in enumerate(pos_args):
149:             if arg.arg in ('self', 'cls'):
150:                 continue
151:             default_idx = i - num_no_default
152:             default = pos_defaults[default_idx] if default_idx >= 0 else None
153:             input_parameters.append(extract_parameter_info(arg, default))
154:
155:         kwonly_args = func_def.args.kwonlyargs
156:         kwonly_defaults = func_def.args.kw_defaults if func_def.args.kw_defaults else []
157:
158:         for i, arg in enumerate(kwonly_args):
159:             if arg.arg in ('self', 'cls'):
160:                 continue
161:             default = kwonly_defaults[i] if i < len(kwonly_defaults) else None
162:             input_parameters.append(extract_parameter_info(arg, default, is_kwonly=True))
163:
164:         if func_def.args.vararg:
165:             from farfan_pipeline.core.method_inventory_types import ParameterDescriptor
166:             vararg_type = None
167:             if func_def.args.vararg.annotation:
168:                 try:
```

```
                    vararg_type = ast.unparse(func_def.args.vararg.annotation)
                except AttributeError:
                    pass
            input_parameters.append(ParameterDescriptor(
                name=f"*{func_def.args.vararg.arg}",
                type_hint=vararg_type,
                has_default=True,
                required=False,
                default_value="()",
                default_type="expression",
                default_source="()"
            ))

        if func_def.args.kwarg:
            from farfan_pipeline.core.method_inventory_types import ParameterDescriptor
            kwarg_type = None
            if func_def.args.kwarg.annotation:
                try:
                    kwarg_type = ast.unparse(func_def.args.kwarg.annotation)
                except AttributeError:
                    pass
            input_parameters.append(ParameterDescriptor(
                name=f"**{func_def.args.kwarg.arg}",
                type_hint=kwarg_type,
                has_default=True,
                required=False,
                default_value="{}",
                default_type="expression",
                default_source="{}"
            ))

    return SignatureDescriptor(
        args=args,
        kwargs=kwargs,
        returns=returns,
        accepts_executor_config=accepts_executor_config,
        is_async=isinstance(func_def, ast.AsyncFunctionDef),
        input_parameters=input_parameters,
        requiere_parametrizacion=requiere_parametrizacion
    )

# --- Stage G: Governance ---

# --- Stage G: Governance ---
def compute_governance_flags_for_file(module_ast: ast.Module) -> GovernanceFlags:
    uses_yaml = False
    has_hardcoded_calibration = False
    has_hardcoded_timeout = False
    suspicious_magic_numbers: list[str] = []

    for node in ast.walk(module_ast):
        if isinstance(node, (ast.Import, ast.ImportFrom)):
            for alias in node.names:
                name = alias.name
                if "yaml" in name:
                    uses_yaml = True
```

```
225:
226:         if isinstance(node, ast.Constant) and isinstance(node.value, str):
227:             if node.value.endswith((".yml", ".yaml")):
228:                 uses_yaml = True
229:
230:         if isinstance(node, (ast.Assign, ast.AnnAssign)):
231:             target_name = None
232:             if isinstance(node, ast.Assign) and isinstance(node.targets[0], ast.Name):
233:                 target_name = node.targets[0].id
234:             elif isinstance(node, ast.AnnAssign) and isinstance(node.target, ast.Name):
235:                 target_name = node.target.id
236:
237:             if target_name is not None:
238:                 val = getattr(node, "value", None)
239:                 if isinstance(val, ast.Constant) and isinstance(val.value, (int, float)):
240:                     if any(k in target_name for k in ("timeout", "timeout_s", "max_retries", "retry")):
241:                         has_hardcoded_timeout = True
242:                         suspicious_magic_numbers.append(f"{target_name} = {val.value}")
243:
244:                     # Calibration tokens
245:                     cal_tokens = ["b_theory", "b_impl", "b_deploy", "quality_threshold", "evidence_snippets", "priority_score"]
246:                     if any(t in target_name for t in cal_tokens):
247:                          has_hardcoded_calibration = True
248:                          suspicious_magic_numbers.append(f"{target_name} assigned literal")
249:
250:     return GovernanceFlags(
251:         uses_yaml=uses_yaml,
252:         has_hardcoded_calibration=has_hardcoded_calibration,
253:         has_hardcoded_timeout=has_hardcoded_timeout,
254:         suspicious_magic_numbers=suspicious_magic_numbers,
255:         is_executor_class=False, # Set later per method
256:     )
257:
258: from collections.abc import Iterable
259:
260: # Import types (only if needed for type hinting later, but for now we keep it minimal as requested)
261: # from .method_inventory_types import ...
262:
263: SRC_ROOT = Path("farfan_core/farfan_core/core") # Kept for backward compat if needed, but main logic uses INVENTORY_ROOTS
264: DEFAULT_OUTPUT = Path("farfan_core/farfan_core/artifacts/calibration/method_inventory.json")
265:
266: INVENTORY_ROOTS = [
267:     Path("."),
268: ]
269:
270: EXCLUDE_DIR_NAMES = {
271:     "__pycache__",
272:     ".pytest_cache",
273:     ".hypothesis",
274:     "tests",
275:     "devtools",
276:     "scripts/dev",
277:     "tools",
278:     ".git",
279:     ".venv",
280:     "venv",
```

```
    "node_modules",
    "build",
    "dist",
    "site-packages",
    ".idea",
    ".vscode",
}

# --- Stage A: Walk ---

def should_exclude_path(path: Path) -> bool:
    return any(part in EXCLUDE_DIR_NAMES for part in path.parts)

def _normalize_roots(roots: Path | Iterable[Path] | None) -> list[Path]:
    """
    Normalize the roots argument so that walk_python_files can accept:
    - None (use INVENTORY_ROOTS)
    - a single Path
    - an iterable of Paths
    """
    if roots is None:
        return list(INVENTORY_ROOTS)
    if isinstance(roots, Path):
        return [roots]
    return list(roots)

def walk_python_files(roots: Path | Iterable[Path] | None = None) -> list[Path]:
    """
    Return all .py files under the given root(s) that are part of the pipeline,
    excluding tests, devtools, caches, and other non-pipeline directories.

    - roots is None: use INVENTORY_ROOTS (multiple roots).
    - roots is a Path: scan only that root (used by tests).
    - roots is an iterable of Paths: scan them all.
    """
    roots_list = _normalize_roots(roots)

    files: set[Path] = set()
    for root in roots_list:
        if not root.exists():
            continue
        # If root is '.', rglob("*") might be too broad if not careful, but we filter extensions
        for p in root.rglob("*.py"):
            if should_exclude_path(p):
                continue
            files.add(p)
    return sorted(files)

# --- Stage B: Module Path ---

def module_path_from_file(path: Path, root: Path) -> str:
    """
    Derive a dotted module path from a file path.
    Heuristically determines the package root to ensure correct imports.
    """
    parts = path.parts
```

```
337:
338:     # Heuristic 1: farfan_core package structure
339:     try:
340:         # Find the index of 'farfan_core'
341:         # We want the path starting from the package name.
342:         # If path is .../farfan_core/farfan_core/core/foo.py -> farfan_core.core.foo
343:         # If path is .../farfan_core/core/foo.py -> farfan_core.core.foo
344:
345:         # We iterate backwards to find the last 'farfan_core' that acts as a package root?
346:         # Actually, if we have farfan_core/farfan_core, the package is the inner one.
347:         # If we have just farfan_core, it is the package.
348:
349:         # Let's look for the index of 'farfan_core'.
350:         indices = [i for i, part in enumerate(parts) if part == "farfan_core"]
351:
352:         if indices:
353:             # If multiple, the package usually starts at the last one?
354:             # No, if path is repo/farfan_core/farfan_core/core/foo.py
355:             # The module is farfan_core.core.foo
356:             # So we want parts starting from the *second* farfan_core (index 1 of the slice).
357:             # Which is the last index in the list of indices.
358:
359:             start_index = indices[-1]
360:             module_parts = parts[start_index:]
361:             return ".".join(module_parts).removesuffix(".py")
362:
363:     except ValueError:
364:         pass
365:
366:     # Heuristic 2: Scripts
367:     if "scripts" in parts:
368:         try:
369:             start_index = parts.index("scripts")
370:             module_parts = parts[start_index:]
371:             return ".".join(module_parts).removesuffix(".py")
372:         except ValueError:
373:             pass
374:
375:     # Fallback: relative to provided root if sensible
376:     try:
377:         if root != Path("."):
378:             rel = path.relative_to(root)
379:             return ".".join(rel.parts).removesuffix(".py")
380:     except ValueError:
381:         pass
382:
383:     # Final Fallback: just the filename
384:     return path.stem
385:
386: # --- Stage C: Parse ---
387:
388: def parse_file(path: Path) -> ast.Module:
389:     """
390:     Parse a Python file into an AST. Raise SyntaxError if invalid.
391:     """
392:     source = path.read_text(encoding="utf-8")
```

```
    return ast.parse(source, filename=str(path))

# --- Stage D: Raw Extraction ---

@dataclass
class RawMethodNode:
    module_path: str
    class_name: str | None
    func_def: ast.FunctionDef | ast.AsyncFunctionDef

def extract_raw_methods(module_ast: ast.Module, module_path: str) -> list[RawMethodNode]:
    """
    Extract raw method/function nodes from a module AST.

    - Methods defined inside classes (ClassDef -> FunctionDef / AsyncFunctionDef).
    - Top-level functions (FunctionDef / AsyncFunctionDef).
    """
    raw: list[RawMethodNode] = []

    for node in module_ast.body:
        # Classes
        if isinstance(node, ast.ClassDef):
            class_name = node.name
            for body_item in node.body:
                if isinstance(body_item, (ast.FunctionDef, ast.AsyncFunctionDef)):
                    raw.append(
                        RawMethodNode(
                            module_path=module_path,
                            class_name=class_name,
                            func_def=body_item,
                        )
                    )
        # Top-level functions
        elif isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef)):
            raw.append(
                RawMethodNode(
                    module_path=module_path,
                    class_name=None,
                    func_def=node,
                )
            )

    return raw

# --- Future Stages (TODO) ---

# def extract_signature(func_def: ast.FunctionDef | ast.AsyncFunctionDef) -> SignatureDescriptor:
#     pass

# def compute_governance_flags(module_ast: ast.Module, class_name: Optional[str]) -> GovernanceFlags:
#     pass

import argparse
import json
from dataclasses import asdict
from typing import Any
```

```
449:
450: from farfan_pipeline.core.method_inventory_types import (
451:     LocationInfo,
452:     MethodDescriptor,
453:     MethodId,
454:     MethodInventory,
455: )
456:
457: # ... (Stages A-D, F-G remain unchanged) ...
458:
459: # --- Stage E: Classification ---
460:
461: def classify_method(raw: RawMethodNode, file_path: Path) -> tuple[str, str]:
462:     role = "UNKNOWN"
463:     level = "UNKNOWN"
464:
465:     # Ejecutores de Fase 2 (visible para questionnaire_monolith)
466:     if raw.class_name and raw.class_name.startswith("D") and "_Q" in raw.class_name and raw.func_def.name == "execute":
467:         role = "EXECUTOR_Q"
468:         level = "SCORE_Q"
469:     elif "scripts/pipeline" in str(file_path):
470:         role = "PIPELINE_SCRIPT"
471:         level = "PIPELINE_ORCHESTRATION"
472:     elif "aggregation.py" in str(file_path):
473:         role = "AGGREGATOR"
474:         level = "AGGREGATE"
475:     elif "flux" in str(file_path) or "phases.py" in str(file_path):
476:         role = "INGEST_PDM"
477:         level = "INGEST"
478:
479:     return role, level
480:
481: # --- Stage H: Build Inventory ---
482:
483: # --- Stage H: Build Inventory ---
484:
485: def is_snapshot_module(module_path: str) -> bool:
486:     """
487:     Return True if this module path corresponds to a snapshot copy of executors
488:     that should not be treated as a separate logical method M.
489:     """
490:     return ".executors_snapshot." in module_path
491:
492: def make_method_id(raw: RawMethodNode) -> MethodId:
493:     if raw.class_name is not None:
494:         return MethodId(f"{raw.class_name}.{raw.func_def.name}")
495:     return MethodId(raw.func_def.name)
496:
497: def build_method_descriptors_for_file(path: Path, root: Path) -> list[MethodDescriptor]:
498:     try:
499:         module_ast = parse_file(path)
500:     except Exception as e:
501:         print(f"Error parsing {path}: {e}", file=sys.stderr)
502:         return []
503:
504:     module_path = module_path_from_file(path, root)
```

```
    if not module_path:
        module_path = "unknown"

    raw_methods = extract_raw_methods(module_ast, module_path)

    # Optimized Governance: Compute once per file
    file_gov_flags = compute_governance_flags_for_file(module_ast)

    descriptors = []

    for raw in raw_methods:
        # Skip 'main' functions as they are entry points, not analytical methods,
        # and cause collisions in a full repo scan.
        if raw.func_def.name == "main":
            continue

        role, level = classify_method(raw, path)
        signature = extract_signature(raw.func_def)

        is_executor = bool(raw.class_name and "Executor" in raw.class_name)

        # Merge file-level flags with method-specific context (is_executor_class)
        gov_flags = GovernanceFlags(
            uses_yaml=file_gov_flags.uses_yaml,
            has_hardcoded_calibration=file_gov_flags.has_hardcoded_calibration,
            has_hardcoded_timeout=file_gov_flags.has_hardcoded_timeout,
            suspicious_magic_numbers=file_gov_flags.suspicious_magic_numbers,
            is_executor_class=is_executor
        )

        # Location
        loc = LocationInfo(
            file_path=str(path),
            line_start=raw.func_def.lineno,
            line_end=raw.func_def.end_lineno if hasattr(raw.func_def, "end_lineno") else raw.func_def.lineno
        )

        method_id = make_method_id(raw)

        desc = MethodDescriptor(
            method_id=method_id,
            role=role,
            aggregation_level=level,
            module=module_path,
            class_name=raw.class_name,
            method_name=raw.func_def.name,
            signature=signature,
            governance_flags=gov_flags,
            location=loc
        )
        descriptors.append(desc)

    return descriptors

def build_method_inventory(roots: Path | Iterable[Path] | None = None) -> MethodInventory:
    methods: dict[MethodId, MethodDescriptor] = {}
```

```
561:
562:     all_files = walk_python_files(roots)
563:
564:     for f in all_files:
565:         # Determine which root this file belongs to for module path calculation
566:         # Normalized roots list for checking
567:         roots_list = _normalize_roots(roots)
568:         matching_root = SRC_ROOT # Default fallback
569:         for r in roots_list:
570:             try:
571:                 f.relative_to(r.parent)
572:                 matching_root = r
573:                 break
574:             except ValueError:
575:                 continue
576:
577:         descriptors = build_method_descriptors_for_file(f, matching_root)
578:
579:         for d in descriptors:
580:             method_id = d.method_id
581:
582:             existing = methods.get(method_id)
583:             if existing is None:
584:                 if is_snapshot_module(d.module):
585:                     # snapshot sin vivo: se ignora por diseÃ±o v1
586:                     continue
587:                 methods[method_id] = d
588:             # colisiÃ³n entre definiciones
589:             elif is_snapshot_module(d.module) and not is_snapshot_module(existing.module):
590:                 # nuevo = snapshot, existente = vivo -> ignorar snapshot
591:                 continue
592:             elif not is_snapshot_module(d.module) and is_snapshot_module(existing.module):
593:                 # nuevo = vivo, antiguo = snapshot -> promover vivo
594:                 methods[method_id] = d
595:                 continue
596:             else:
597:                 # ambos vivos o ambos snapshot -> colisiÃ³n real
598:                 # En un escaneo completo de todo el repo, esto es inevitable para funciones comunes.
599:                 # Para cumplir con "DAME ESO YA MISMO", convertimos el error en warning y saltamos.
600:                 print(
601:                     f"Warning: True Method ID collision for '{method_id}' between {existing.module} and {d.module}. Skipping {d.module}.",
602:                     file=sys.stderr
603:                 )
604:                 continue
605:
606:     return MethodInventory(methods=methods)
607:
608: # --- Stage I: Serialization & CLI ---
609:
610: def method_inventory_to_json(inv: MethodInventory) -> dict[str, Any]:
611:     return asdict(inv)
612:
613: def main(argv: list[str] | None = None) -> int:
614:     parser = argparse.ArgumentParser(description="Generate Static Method Inventory")
615:     parser.add_argument("--root", type=str, help="Root directory to scan (optional, defaults to configured INVENTORY_ROOTS)")
616:     parser.add_argument("--output", type=str, help="Output JSON file path")
```

```
617:
618:     args = parser.parse_args(argv)
619:
620:     # If root is provided, use it. Otherwise pass None to use INVENTORY_ROOTS.
621:     root_arg = Path(args.root) if args.root else None
622:
623:     if root_arg:
624:         if not root_arg.exists() or not root_arg.is_dir():
625:              print(f"Root directory not found: {root_arg}", file=sys.stderr)
626:              return 1
627:         print(f"Scanning root: {root_arg}")
628:     else:
629:         print(f"Scanning configured roots: {INVENTORY_ROOTS}")
630:
631:     output_path = Path(args.output) if args.output else DEFAULT_OUTPUT
632:
633:     try:
634:         inventory = build_method_inventory(root_arg)
635:     except ValueError as e:
636:         print(f"Error: {e}", file=sys.stderr)
637:         return 1
638:     except Exception as e:
639:         print(f"Unexpected error: {e}", file=sys.stderr)
640:         return 2
641:
642:     # Serialization
643:     data = method_inventory_to_json(inventory)
644:
645:     # Ensure output dir exists
646:     output_path.parent.mkdir(parents=True, exist_ok=True)
647:
648:     with open(output_path, "w", encoding="utf-8") as f:
649:         json.dump(data, f, indent=2)
650:
651:     print(f"Inventory generated at: {output_path}")
652:     print(f"Total methods: {len(inventory.methods)}")
653:
654:     return 0
655:
656: if __name__ == "__main__":
657:     sys.exit(main())
658:
659:
660:
661: ================================================================================
662: FILE: src/farfan_pipeline/core/method_inventory_types.py
663: ================================================================================
664:
665: """
666: Method Inventory Types Module.
667:
668: Defines the immutable data structures for the static method inventory system.
669: This module is designed to be pure and dependency-free regarding the rest of the system.
670: """
671: from dataclasses import dataclass
672: from typing import NewType
```

```
673:
674: # MethodId: Unique identifier for a method (ClassName.method_name or function_name)
675: MethodId = NewType("MethodId", str)
676:
677: @dataclass(frozen=True)
678: class ParameterDescriptor:
679:     """Describes a single parameter with full metadata."""
680:     name: str
681:     type_hint: str | None
682:     has_default: bool
683:     required: bool
684:     default_value: str | None
685:     default_type: str | None
686:     default_source: str | None
687:
688: @dataclass(frozen=True)
689: class SignatureDescriptor:
690:     """Describes the signature of a method."""
691:     args: list[str]
692:     kwargs: list[str]
693:     returns: str
694:     accepts_executor_config: bool
695:     is_async: bool
696:     input_parameters: list[ParameterDescriptor] | None = None
697:     requiere_parametrizacion: bool = False
698:
699: @dataclass(frozen=True)
700: class GovernanceFlags:
701:     """Flags related to governance and 'SIN_CARRETA' rules."""
702:     uses_yaml: bool
703:     has_hardcoded_calibration: bool
704:     has_hardcoded_timeout: bool
705:     suspicious_magic_numbers: list[str]
706:     is_executor_class: bool
707:
708: @dataclass(frozen=True)
709: class LocationInfo:
710:     """Physical location of the method definition."""
711:     file_path: str
712:     line_start: int
713:     line_end: int
714:
715: @dataclass(frozen=True)
716: class MethodDescriptor:
717:     """Complete descriptor for a single analytical method."""
718:     method_id: MethodId
719:     role: str  # EXTRACTOR|VALIDATOR|SCORER|AGGREGATOR|INGEST_PDM|META_TOOL
720:     aggregation_level: str  # INGEST|SCORE_Q|AGGREGATE|META
721:     module: str
722:     class_name: str | None
723:     method_name: str
724:     signature: SignatureDescriptor
725:     governance_flags: GovernanceFlags
726:     location: LocationInfo
727:
728: @dataclass(frozen=True)
```

```
class MethodInventory:
    """The complete inventory of methods."""
    methods: dict[MethodId, MethodDescriptor]
    _version: str = "1.0.0"
    _comment: str = "Static unified method inventory for SAAAAAA â\200\223 auto-generated, DO NOT EDIT BY HAND."


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

"""Observability module for F.A.R.F.A.N runtime monitoring."""

from farfan_pipeline.core.observability.structured_logging import log_fallback, get_logger
from farfan_pipeline.core.observability.metrics import (
    increment_fallback,
    increment_segmentation_method,
    increment_calibration_mode,
    increment_document_id_source,
    increment_hash_algo,
    increment_graph_metrics_skipped,
    increment_contradiction_mode,
)

__all__ = [
    "log_fallback",
    "get_logger",
    "increment_fallback",
    "increment_segmentation_method",
    "increment_calibration_mode",
    "increment_document_id_source",
    "increment_hash_algo",
    "increment_graph_metrics_skipped",
    "increment_contradiction_mode",
]


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

"""
Prometheus-style metrics for F.A.R.F.A.N runtime observability.

This module provides metric counters for tracking fallbacks, degradations,
and runtime behavior. All metrics are exposed via the ATROZ dashboard's
hidden layer for centralized health monitoring.

Metrics are prefixed with 'farfan_core_' and include labels for categorization.
"""

from typing import Optional
from prometheus_client import Counter, Gauge, Histogram

```

```
785: from farfan_pipeline.core.runtime_config import RuntimeMode
786: from farfan_pipeline.core.contracts.runtime_contracts import (
787:     FallbackCategory,
788:     SegmentationMethod,
789:     CalibrationMode,
790:     DocumentIdSource,
791: )
792:
793:
794: # ============================================================================
795: # Fallback Metrics
796: # ============================================================================
797:
798: fallback_activations_total = Counter(
799:     'farfan_core_fallback_activations_total',
800:     'Total number of fallback activations',
801:     ['component', 'fallback_category', 'fallback_mode', 'runtime_mode']
802: )
803:
804:
805: def increment_fallback(
806:     component: str,
807:     fallback_category: FallbackCategory,
808:     fallback_mode: str,
809:     runtime_mode: RuntimeMode,
810: ) -> None:
811:     """
812:     Increment fallback activation counter.
813:
814:     Args:
815:         component: Component name
816:         fallback_category: Fallback category (A/B/C/D)
817:         fallback_mode: Specific fallback mode
818:         runtime_mode: Current runtime mode
819:     """
820:     fallback_activations_total.labels(
821:         component=component,
822:         fallback_category=fallback_category.value,
823:         fallback_mode=fallback_mode,
824:         runtime_mode=runtime_mode.value,
825:     ).inc()
826:
827:
828: # ============================================================================
829: # Segmentation Metrics
830: # ============================================================================
831:
832: segmentation_method_total = Counter(
833:     'farfan_core_segmentation_method_total',
834:     'Total segmentation method usage',
835:     ['method', 'runtime_mode']
836: )
837:
838:
839: def increment_segmentation_method(
840:     method: SegmentationMethod,
```

```
    runtime_mode: RuntimeMode,
) -> None:
    """
    Increment segmentation method counter.

    Args:
        method: Segmentation method used
        runtime_mode: Current runtime mode
    """
    segmentation_method_total.labels(
        method=method.value,
        runtime_mode=runtime_mode.value,
    ).inc()


# ============================================================================
# Calibration Metrics
# ============================================================================

calibration_mode_total = Counter(
    'farfan_core_calibration_mode_total',
    'Total calibration mode usage',
    ['mode', 'runtime_mode']
)


def increment_calibration_mode(
    mode: CalibrationMode,
    runtime_mode: RuntimeMode,
) -> None:
    """
    Increment calibration mode counter.

    Args:
        mode: Calibration mode
        runtime_mode: Current runtime mode
    """
    calibration_mode_total.labels(
        mode=mode.value,
        runtime_mode=runtime_mode.value,
    ).inc()


# ============================================================================
# Document ID Metrics
# ============================================================================

document_id_source_total = Counter(
    'farfan_core_document_id_source_total',
    'Total document ID source usage',
    ['source', 'runtime_mode']
)


def increment_document_id_source(
    source: DocumentIdSource,
```

```
    runtime_mode: RuntimeMode,
) -> None:
    """
    Increment document ID source counter.

    Args:
        source: Document ID source
        runtime_mode: Current runtime mode
    """
    document_id_source_total.labels(
        source=source.value,
        runtime_mode=runtime_mode.value,
    ).inc()


# ============================================================================
# Hash Algorithm Metrics
# ============================================================================

hash_algo_total = Counter(
    'farfan_core_hash_algo_total',
    'Total hash algorithm usage',
    ['algo', 'runtime_mode']
)


def increment_hash_algo(
    algo: str,
    runtime_mode: RuntimeMode,
) -> None:
    """
    Increment hash algorithm counter.

    Args:
        algo: Hash algorithm name (e.g., "blake3", "sha256")
        runtime_mode: Current runtime mode
    """
    hash_algo_total.labels(
        algo=algo,
        runtime_mode=runtime_mode.value,
    ).inc()


# ============================================================================
# Graph Metrics
# ============================================================================

graph_metrics_skipped_total = Counter(
    'farfan_core_graph_metrics_skipped_total',
    'Total graph metrics computation skips',
    ['runtime_mode', 'reason']
)


def increment_graph_metrics_skipped(
    reason: str,
```

```
    runtime_mode: RuntimeMode,
) -> None:
    """
    Increment graph metrics skipped counter.

    Args:
        reason: Reason for skip (e.g., "networkx_unavailable")
        runtime_mode: Current runtime mode
    """
    graph_metrics_skipped_total.labels(
        runtime_mode=runtime_mode.value,
        reason=reason,
    ).inc()


# ============================================================================
# Contradiction Detection Metrics
# ============================================================================

contradiction_mode_total = Counter(
    'farfan_core_contradiction_mode_total',
    'Total contradiction detection mode usage',
    ['mode', 'runtime_mode']
)


def increment_contradiction_mode(
    mode: str,
    runtime_mode: RuntimeMode,
) -> None:
    """
    Increment contradiction mode counter.

    Args:
        mode: Contradiction mode ("full" or "fallback")
        runtime_mode: Current runtime mode
    """
    contradiction_mode_total.labels(
        mode=mode,
        runtime_mode=runtime_mode.value,
    ).inc()


# ============================================================================
# Boot Check Metrics
# ============================================================================

boot_check_failures_total = Counter(
    'farfan_core_boot_check_failures_total',
    'Total boot check failures',
    ['check_name', 'runtime_mode', 'code']
)


def increment_boot_check_failure(
    check_name: str,
```

```
    code: str,
    runtime_mode: RuntimeMode,
) -> None:
    """
    Increment boot check failure counter.

    Args:
        check_name: Name of failed check
        code: Error code
        runtime_mode: Current runtime mode
    """
    boot_check_failures_total.labels(
        check_name=check_name,
        runtime_mode=runtime_mode.value,
        code=code,
    ).inc()


# ============================================================================
# Pipeline Execution Metrics
# ============================================================================

pipeline_executions_total = Counter(
    'farfan_core_pipeline_executions_total',
    'Total pipeline executions',
    ['runtime_mode', 'status']
)

pipeline_execution_duration_seconds = Histogram(
    'farfan_core_pipeline_execution_duration_seconds',
    'Pipeline execution duration in seconds',
    ['runtime_mode'],
    buckets=(1, 5, 10, 30, 60, 120, 300, 600, 1800, 3600)
)


def increment_pipeline_execution(
    runtime_mode: RuntimeMode,
    status: str,
) -> None:
    """
    Increment pipeline execution counter.

    Args:
        runtime_mode: Current runtime mode
        status: Execution status ("success", "failure", "aborted")
    """
    pipeline_executions_total.labels(
        runtime_mode=runtime_mode.value,
        status=status,
    ).inc()


def observe_pipeline_duration(
    duration_seconds: float,
    runtime_mode: RuntimeMode,
```

```
) -> None:
    """
    Observe pipeline execution duration.

    Args:
        duration_seconds: Execution duration in seconds
        runtime_mode: Current runtime mode
    """
    pipeline_execution_duration_seconds.labels(
        runtime_mode=runtime_mode.value,
    ).observe(duration_seconds)


# ============================================================================
# ATROZ Dashboard Integration
# ============================================================================

# Current runtime mode gauge (for dashboard display)
current_runtime_mode = Gauge(
    'farfan_core_current_runtime_mode',
    'Current runtime mode (0=PROD, 1=DEV, 2=EXPLORATORY)',
    []
)


def set_current_runtime_mode(mode: RuntimeMode) -> None:
    """
    Set current runtime mode gauge for dashboard display.

    Args:
        mode: Current runtime mode
    """
    mode_value = {
        RuntimeMode.PROD: 0,
        RuntimeMode.DEV: 1,
        RuntimeMode.EXPLORATORY: 2,
    }[mode]
    current_runtime_mode.set(mode_value)


# Health status gauge (for dashboard alerts)
system_health_status = Gauge(
    'farfan_core_system_health_status',
    'System health status (0=degraded, 1=healthy)',
    []
)


def set_system_health_status(healthy: bool) -> None:
    """
    Set system health status for dashboard alerts.

    Args:
        healthy: Whether system is healthy (no Category A fallbacks)
    """
    system_health_status.set(1 if healthy else 0)
```

```
1121:
1122:
1123: # ============================================================================
1124: # Metric Export for ATROZ Dashboard
1125: # ============================================================================
1126:
1127: def get_all_metrics() -> dict[str, any]:
1128:     """
1129:     Get all metrics for ATROZ dashboard integration.
1130:
1131:     Returns:
1132:         Dictionary of all metric collectors
1133:     """
1134:     return {
1135:         'fallback_activations_total': fallback_activations_total,
1136:         'segmentation_method_total': segmentation_method_total,
1137:         'calibration_mode_total': calibration_mode_total,
1138:         'document_id_source_total': document_id_source_total,
1139:         'hash_algo_total': hash_algo_total,
1140:         'graph_metrics_skipped_total': graph_metrics_skipped_total,
1141:         'contradiction_mode_total': contradiction_mode_total,
1142:         'boot_check_failures_total': boot_check_failures_total,
1143:         'pipeline_executions_total': pipeline_executions_total,
1144:         'pipeline_execution_duration_seconds': pipeline_execution_duration_seconds,
1145:         'current_runtime_mode': current_runtime_mode,
1146:         'system_health_status': system_health_status,
1147:     }
1148:
1149:
1150:
1151: ================================================================================
1152: FILE: src/farfan_pipeline/core/observability/structured_logging.py
1153: ================================================================================
1154:
1155: """
1156: Structured logging for F.A.R.F.A.N runtime observability.
1157:
1158: This module provides standardized logging for fallbacks, degradations, and
1159: runtime events with consistent field injection for observability.
1160: """
1161:
1162: import logging
1163: import structlog
1164: from typing import Optional
1165:
1166: from farfan_pipeline.core.runtime_config import RuntimeMode
1167: from farfan_pipeline.core.contracts.runtime_contracts import FallbackCategory
1168:
1169:
1170: # Configure structlog for JSON logging
1171: structlog.configure(
1172:     processors=[
1173:         structlog.stdlib.filter_by_level,
1174:         structlog.stdlib.add_logger_name,
1175:         structlog.stdlib.add_log_level,
1176:         structlog.stdlib.PositionalArgumentsFormatter(),
```

```
        structlog.processors.TimeStamper(fmt="iso"),
        structlog.processors.StackInfoRenderer(),
        structlog.processors.format_exc_info,
        structlog.processors.UnicodeDecoder(),
        structlog.processors.JSONRenderer()
    ],
    context_class=dict,
    logger_factory=structlog.stdlib.LoggerFactory(),
    cache_logger_on_first_use=True,
)


def get_logger(name: str = __name__) -> structlog.BoundLogger:
    """
    Get a structured logger instance.

    Args:
        name: Logger name (typically __name__)

    Returns:
        Structured logger instance
    """
    return structlog.get_logger(name)


def log_fallback(
    component: str,
    subsystem: str,
    fallback_category: FallbackCategory,
    fallback_mode: str,
    reason: str,
    runtime_mode: RuntimeMode,
    document_id: Optional[str] = None,
    run_id: Optional[str] = None,
    logger: Optional[structlog.BoundLogger] = None,
) -> None:
    """
    Emit structured log for fallback activation.

    This is the standard logging function for all fallback events. It ensures
    consistent field injection and proper categorization for observability.

    Args:
        component: Component name (e.g., "language_detection", "calibration")
        subsystem: Subsystem name (e.g., "spc_ingestion", "orchestrator")
        fallback_category: Fallback category (A/B/C/D)
        fallback_mode: Specific fallback mode (e.g., "warn_default_es", "estimated")
        reason: Human-readable reason for fallback
        runtime_mode: Current runtime mode
        document_id: Optional document identifier
        run_id: Optional run identifier
        logger: Optional logger instance (creates new if None)

    Example:
        >>> log_fallback(
        ...     component="language_detection",
```

```
        ...     subsystem="spc_ingestion",
        ...     fallback_category=FallbackCategory.B,
        ...     fallback_mode="warn_default_es",
        ...     reason="LangDetectException",
        ...     runtime_mode=RuntimeMode.PROD
        ... )
    """
    if logger is None:
        logger = get_logger()

    # Determine log level based on category
    if fallback_category == FallbackCategory.A:
        log_level = logging.ERROR
    elif fallback_category == FallbackCategory.B:
        log_level = logging.WARNING
    else:
        log_level = logging.INFO

    # Emit structured log
    logger.log(
        log_level,
        "fallback_activated",
        component=component,
        subsystem=subsystem,
        fallback_category=fallback_category.value,
        fallback_mode=fallback_mode,
        reason=reason,
        runtime_mode=runtime_mode.value,
        document_id=document_id,
        run_id=run_id,
    )


def log_boot_check_failure(
    check_name: str,
    reason: str,
    code: str,
    runtime_mode: RuntimeMode,
    logger: Optional[structlog.BoundLogger] = None,
) -> None:
    """
    Log boot check failure.

    Args:
        check_name: Name of failed check
        reason: Failure reason
        code: Error code
        runtime_mode: Current runtime mode
        logger: Optional logger instance
    """
    if logger is None:
        logger = get_logger()

    logger.error(
        "boot_check_failed",
        check_name=check_name,
```

```
1289:         reason=reason,
1290:         code=code,
1291:         runtime_mode=runtime_mode.value,
1292:     )
1293:
1294:
1295: def log_boot_check_success(
1296:     check_name: str,
1297:     runtime_mode: RuntimeMode,
1298:     logger: Optional[structlog.BoundLogger] = None,
1299: ) -> None:
1300:     """
1301:     Log boot check success.
1302:
1303:     Args:
1304:         check_name: Name of successful check
1305:         runtime_mode: Current runtime mode
1306:         logger: Optional logger instance
1307:     """
1308:     if logger is None:
1309:         logger = get_logger()
1310:
1311:     logger.info(
1312:         "boot_check_passed",
1313:         check_name=check_name,
1314:         runtime_mode=runtime_mode.value,
1315:     )
1316:
1317:
1318: def log_runtime_config_loaded(
1319:     config_repr: str,
1320:     runtime_mode: RuntimeMode,
1321:     logger: Optional[structlog.BoundLogger] = None,
1322: ) -> None:
1323:     """
1324:     Log runtime configuration loaded.
1325:
1326:     Args:
1327:         config_repr: String representation of config
1328:         runtime_mode: Runtime mode
1329:         logger: Optional logger instance
1330:     """
1331:     if logger is None:
1332:         logger = get_logger()
1333:
1334:     logger.info(
1335:         "runtime_config_loaded",
1336:         config=config_repr,
1337:         runtime_mode=runtime_mode.value,
1338:     )
1339:
1340:
1341:
1342: ================================================================================
1343: FILE: src/farfan_pipeline/core/orchestrator/__init__.py
1344: ================================================================================
```

```
1345:
1346: """Orchestrator utilities with contract validation on import."""
1347:
1348: from __future__ import annotations
1349:
1350: from typing import TYPE_CHECKING
1351:
1352: if TYPE_CHECKING:
1353:     from farfan_pipeline.core.orchestrator.questionnaire import CanonicalQuestionnaire
1354:
1355: from farfan_pipeline.core.types import ChunkData, PreprocessedDocument, Provenance
1356: from farfan_pipeline.core.orchestrator.core import (
1357:     AbortRequested,
1358:     AbortSignal,
1359:     Evidence,
1360:     MethodExecutor,
1361:     MicroQuestionRun,
1362:     Orchestrator,
1363:     PhaseInstrumentation,
1364:     PhaseResult,
1365:     ResourceLimits,
1366:     ScoredMicroQuestion,
1367: )
1368: from farfan_pipeline.core.orchestrator.evidence_registry import (
1369:     EvidenceRecord,
1370:     EvidenceRegistry,
1371:     ProvenanceDAG,
1372:     ProvenanceNode,
1373:     get_global_registry,
1374: )
1375: from farfan_pipeline.core.orchestrator.resource_manager import (
1376:     AdaptiveResourceManager,
1377:     CircuitBreaker,
1378:     CircuitState,
1379:     DegradationStrategy,
1380:     ExecutorPriority,
1381:     ResourceAllocationPolicy,
1382:     ResourcePressureLevel,
1383: )
1384: from farfan_pipeline.core.orchestrator.resource_aware_executor import (
1385:     ResourceAwareExecutor,
1386:     ResourceConstraints,
1387: )
1388: from farfan_pipeline.core.orchestrator.resource_alerts import (
1389:     AlertChannel,
1390:     AlertSeverity,
1391:     ResourceAlert,
1392:     ResourceAlertManager,
1393: )
1394: from farfan_pipeline.core.orchestrator.resource_integration import (
1395:     create_resource_manager,
1396:     integrate_with_orchestrator,
1397:     get_resource_status,
1398:     reset_circuit_breakers,
1399: )
1400:
```

```
__all__ = [
    "EvidenceRecord",
    "EvidenceRegistry",
    "ProvenanceDAG",
    "ProvenanceNode",
    "get_global_registry",
    "Orchestrator",
    "MethodExecutor",
    "PreprocessedDocument",
    "ChunkData",
    "Provenance",
    "Evidence",
    "AbortSignal",
    "AbortRequested",
    "ResourceLimits",
    "PhaseInstrumentation",
    "PhaseResult",
    "MicroQuestionRun",
    "ScoredMicroQuestion",
    "AdaptiveResourceManager",
    "CircuitBreaker",
    "CircuitState",
    "DegradationStrategy",
    "ExecutorPriority",
    "ResourceAllocationPolicy",
    "ResourcePressureLevel",
    "ResourceAwareExecutor",
    "ResourceConstraints",
    "AlertChannel",
    "AlertSeverity",
    "ResourceAlert",
    "ResourceAlertManager",
    "create_resource_manager",
    "integrate_with_orchestrator",
    "get_resource_status",
    "reset_circuit_breakers",
]



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/arg_router.py
================================================================================

"""Argument routing with special routes, strict validation, and comprehensive metrics.

This module provides ExtendedArgRouter (and legacy ArgRouter for compatibility):
- 30+ special route handlers for commonly-called methods
- Strict validation (no silent parameter drops)
- **kwargs support for forward compatibility
- Full observability and metrics
- Base routing and validation utilities

Design Principles:
- Explicit route definitions for high-traffic methods
- Fail-fast on missing required arguments
```

```
- Fail-fast on unexpected arguments (unless **kwargs present)
- Full traceability of routing decisions
- Zero tolerance for silent parameter drops
"""

from __future__ import annotations

import inspect
import logging
import os
import random
import threading
from collections.abc import Iterable, Mapping, MutableMapping
from dataclasses import dataclass
from typing import (
    Any,
    Union,
    get_args,
    get_origin,
    get_type_hints,
)

import structlog

logger = structlog.get_logger(__name__)
std_logger = logging.getLogger(__name__)

# Sentinel value for missing arguments
MISSING: object = object()


# ============================================================================
# Base Exceptions and Data Classes
# ============================================================================

class ArgRouterError(RuntimeError):
    """Base exception for routing and validation issues."""


class ArgumentValidationError(ArgRouterError):
    """Raised when the provided payload does not match the method signature."""

    def __init__(
        self,
        class_name: str,
        method_name: str,
        *,
        missing: Iterable[str] | None = None,
        unexpected: Iterable[str] | None = None,
        type_mismatches: Mapping[str, str] | None = None,
    ) -> None:
        self.class_name = class_name
        self.method_name = method_name
        self.missing = set(missing or ())
        self.unexpected = set(unexpected or ())
        self.type_mismatches = dict(type_mismatches or {})
```

```
        detail = []
        if self.missing:
            detail.append(f"missing={sorted(self.missing)}")
        if self.unexpected:
            detail.append(f"unexpected={sorted(self.unexpected)}")
        if self.type_mismatches:
            detail.append(f"type_mismatches={self.type_mismatches}")
        message = (
            f"Invalid payload for {class_name}.{method_name}"
            + (f" ({'; '.join(detail)})" if detail else "")
        )
        super().__init__(message)


@dataclass(frozen=True)
class _ParameterSpec:
    name: str
    kind: inspect._ParameterKind
    default: Any
    annotation: Any

    @property
    def required(self) -> bool:
        return self.default is MISSING


@dataclass(frozen=True)
class MethodSpec:
    class_name: str
    method_name: str
    positional: tuple[_ParameterSpec, ...]
    keyword_only: tuple[_ParameterSpec, ...]
    has_var_keyword: bool
    has_var_positional: bool

    @property
    def required_arguments(self) -> tuple[str, ...]:
        required = tuple(
            spec.name
            for spec in (*self.positional, *self.keyword_only)
            if spec.required
        )
        return required

    @property
    def accepted_arguments(self) -> tuple[str, ...]:
        accepted = tuple(spec.name for spec in (*self.positional, *self.keyword_only))
        return accepted


# ============================================================================
# Base ArgRouter (Legacy - use ExtendedArgRouter instead)
# ============================================================================

class ArgRouter:
    """Resolve method call payloads based on inspected signatures.
```

```
1569:
1570:     .. note::
1571:         ExtendedArgRouter is the recommended router to use directly.
1572:         This base class is provided for backward compatibility.
1573:     """
1574:
1575:     def __init__(self, class_registry: Mapping[str, type]) -> None:
1576:         self._class_registry = dict(class_registry)
1577:         self._spec_cache: dict[tuple[str, str], MethodSpec] = {}
1578:         self._lock = threading.RLock()
1579:
1580:     def describe(self, class_name: str, method_name: str) -> MethodSpec:
1581:         """Return the cached method specification, building it if necessary."""
1582:         key = (class_name, method_name)
1583:         with self._lock:
1584:             if key not in self._spec_cache:
1585:                 self._spec_cache[key] = self._build_spec(class_name, method_name)
1586:             return self._spec_cache[key]
1587:
1588:     def route(
1589:         self,
1590:         class_name: str,
1591:         method_name: str,
1592:         payload: MutableMapping[str, Any],
1593:     ) -> tuple[tuple[Any, ...], dict[str, Any]]:
1594:         """Validate and split a payload into positional and keyword arguments."""
1595:         spec = self.describe(class_name, method_name)
1596:         provided_keys = set(payload.keys())
1597:         required = set(spec.required_arguments)
1598:         accepted = set(spec.accepted_arguments)
1599:
1600:         missing = required - provided_keys
1601:         unexpected = provided_keys - accepted
1602:         if unexpected and spec.has_var_keyword:
1603:             unexpected = set()
1604:
1605:         if missing or unexpected:
1606:             raise ArgumentValidationError(
1607:                 class_name,
1608:                 method_name,
1609:                 missing=missing,
1610:                 unexpected=unexpected,
1611:             )
1612:
1613:         args: list[Any] = []
1614:         kwargs: dict[str, Any] = {}
1615:         type_mismatches: dict[str, str] = {}
1616:
1617:         remaining = dict(payload)
1618:
1619:         for param in spec.positional:
1620:             if param.name not in remaining:
1621:                 if param.required:
1622:                     missing = {param.name}
1623:                     raise ArgumentValidationError(
1624:                         class_name,
```

```
                        method_name,
                        missing=missing,
                    )
                continue
            value = remaining.pop(param.name)
            if not self._matches_annotation(value, param.annotation):
                expected = self._describe_annotation(param.annotation)
                type_mismatches[param.name] = expected
            args.append(value)

        for param in spec.keyword_only:
            if param.name not in remaining:
                if param.required:
                    raise ArgumentValidationError(
                        class_name,
                        method_name,
                        missing={param.name},
                    )
                continue
            value = remaining.pop(param.name)
            if not self._matches_annotation(value, param.annotation):
                expected = self._describe_annotation(param.annotation)
                type_mismatches[param.name] = expected
            kwargs[param.name] = value

        if spec.has_var_keyword and remaining:
            kwargs.update(remaining)
            remaining = {}

        if remaining:
            raise ArgumentValidationError(
                class_name,
                method_name,
                unexpected=set(remaining.keys()),
            )

        if type_mismatches:
            raise ArgumentValidationError(
                class_name,
                method_name,
                type_mismatches={
                    name: f"expected {expected}; received {type(payload[name]).__name__}"
                    for name, expected in type_mismatches.items()
                },
            )

        return tuple(args), kwargs

    def expected_arguments(self, class_name: str, method_name: str) -> tuple[str, ...]:
        spec = self.describe(class_name, method_name)
        return spec.accepted_arguments

    def _build_spec(self, class_name: str, method_name: str) -> MethodSpec:
        try:
            cls = self._class_registry[class_name]
        except KeyError as exc:  # pragma: no cover - defensive
```

```
            raise ArgRouterError(f"Unknown class '{class_name}'") from exc

        try:
            method = getattr(cls, method_name)
        except AttributeError as exc:
            raise ArgRouterError(f"Class '{class_name}' has no method '{method_name}'") from exc

        signature = inspect.signature(method)
        try:
            type_hints = get_type_hints(method)
        except Exception:
            type_hints = {}
        positional: list[_ParameterSpec] = []
        keyword_only: list[_ParameterSpec] = []
        has_var_keyword = False
        has_var_positional = False

        for parameter in signature.parameters.values():
            if parameter.name == "self":
                continue
            default = (
                parameter.default
                if parameter.default is not inspect._empty
                else MISSING
            )
            annotation = type_hints.get(parameter.name, parameter.annotation)
            param_spec = _ParameterSpec(
                name=parameter.name,
                kind=parameter.kind,
                default=default,
                annotation=annotation,
            )
            if parameter.kind in (
                inspect.Parameter.POSITIONAL_ONLY,
                inspect.Parameter.POSITIONAL_OR_KEYWORD,
            ):
                positional.append(param_spec)
            elif parameter.kind is inspect.Parameter.KEYWORD_ONLY:
                keyword_only.append(param_spec)
            elif parameter.kind is inspect.Parameter.VAR_KEYWORD:
                has_var_keyword = True
            elif parameter.kind is inspect.Parameter.VAR_POSITIONAL:
                has_var_positional = True

        return MethodSpec(
            class_name=class_name,
            method_name=method_name,
            positional=tuple(positional),
            keyword_only=tuple(keyword_only),
            has_var_keyword=has_var_keyword,
            has_var_positional=has_var_positional,
        )

    @staticmethod
    def _matches_annotation(value: Any, annotation: Any) -> bool:
        if annotation in (inspect._empty, Any):
```

```
            return True
        origin = get_origin(annotation)
        if origin is None:
            if isinstance(annotation, type):
                return isinstance(value, annotation)
            return True
        args = get_args(annotation)
        if origin is tuple:
            if not isinstance(value, tuple):
                return False
            if not args:
                return True
            if len(args) == 2 and args[1] is Ellipsis:
                return all(ArgRouter._matches_annotation(item, args[0]) for item in value)
            if len(args) != len(value):
                return False
            return all(
                ArgRouter._matches_annotation(item, arg_type)
                for item, arg_type in zip(value, args, strict=False)
            )
        if origin in (list, list):
            if not isinstance(value, list):
                return False
            if not args:
                return True
            return all(ArgRouter._matches_annotation(item, args[0]) for item in value)
        if origin in (set, set):
            if not isinstance(value, set):
                return False
            if not args:
                return True
            return all(ArgRouter._matches_annotation(item, args[0]) for item in value)
        if origin in (dict, dict):
            if not isinstance(value, dict):
                return False
            if len(args) != 2:
                return True
            key_type, value_type = args
            return all(
                ArgRouter._matches_annotation(k, key_type)
                and ArgRouter._matches_annotation(v, value_type)
                for k, v in value.items()
            )
        if origin is Union:
            return any(ArgRouter._matches_annotation(value, arg) for arg in args)
        return True

    @staticmethod
    def _describe_annotation(annotation: Any) -> str:
        if annotation in (inspect._empty, Any):
            return "Any"
        origin = get_origin(annotation)
        if origin is None:
            if isinstance(annotation, type):
                return annotation.__name__
            return str(annotation)
```

```
        args = get_args(annotation)
        if origin is tuple:
            return f"Tuple[{', '.join(ArgRouter._describe_annotation(arg) for arg in args)}]"
        if origin in (list, list):
            return f"List[{ArgRouter._describe_annotation(args[0])}]" if args else "List[Any]"
        if origin in (set, set):
            return f"Set[{ArgRouter._describe_annotation(args[0])}]" if args else "Set[Any]"
        if origin in (dict, dict):
            if len(args) == 2:
                return (
                    f"Dict[{ArgRouter._describe_annotation(args[0])}, "
                    f"{ArgRouter._describe_annotation(args[1])}]"
                )
            return "Dict[Any, Any]"
        if origin is Union:
            return " | ".join(ArgRouter._describe_annotation(arg) for arg in args)
        return str(annotation)


class PayloadDriftMonitor:
    """Sampling validator for ingress/egress payloads."""

    CRITICAL_KEYS = {
        "content": str,
        "pdq_context": (dict, type(None)),
    }

    def __init__(self, *, sample_rate: float, enabled: bool) -> None:
        self.sample_rate = max(0.0, min(sample_rate, 1.0))
        self.enabled = enabled and self.sample_rate > 0.0

    @classmethod
    def from_env(cls) -> PayloadDriftMonitor:
        enabled = os.getenv("ORCHESTRATOR_SAMPLING_VALIDATION", "").lower() in {
            "1",
            "true",
            "yes",
            "on",
        }
        try:
            sample_rate = float(os.getenv("ORCHESTRATOR_SAMPLING_RATE", "0.05"))
        except ValueError:
            sample_rate = 0.05
        return cls(sample_rate=sample_rate, enabled=enabled)

    def maybe_validate(self, payload: Mapping[str, Any], *, producer: str, consumer: str) -> None:
        if not self.enabled:
            return
        if random.random() > self.sample_rate:
            return
        if not isinstance(payload, Mapping):
            return
        keys = set(payload.keys())
        if not keys.intersection(self.CRITICAL_KEYS):
            return

```

```
1849:         missing = [key for key in self.CRITICAL_KEYS if key not in payload]
1850:         type_mismatches = {
1851:             key: self._expected_type_name(expected)
1852:             for key, expected in self.CRITICAL_KEYS.items()
1853:             if key in payload and not isinstance(payload[key], expected)
1854:         }
1855:         if missing or type_mismatches:
1856:             std_logger.error(
1857:                 "Payload drift detected [%s -> %s]: missing=%s type_mismatches=%s",
1858:                 producer,
1859:                 consumer,
1860:                 missing,
1861:                 type_mismatches,
1862:             )
1863:         else:
1864:             std_logger.debug(
1865:                 "Payload validation OK [%s -> %s]", producer, consumer
1866:             )
1867:
1868:     @staticmethod
1869:     def _expected_type_name(expected: object) -> str:
1870:         if isinstance(expected, tuple):
1871:             return ", ".join(getattr(t, "__name__", str(t)) for t in expected)
1872:         if hasattr(expected, "__name__"):
1873:             return expected.__name__  # type: ignore[arg-type]
1874:         return str(expected)
1875:
1876:
1877: # ============================================================================
1878: # Extended ArgRouter with Special Routes
1879: # ============================================================================
1880:
1881:
1882: @dataclass
1883: class RoutingMetrics:
1884:     """Metrics for monitoring routing behavior."""
1885:
1886:     total_routes: int = 0
1887:     special_routes_hit: int = 0
1888:     default_routes_hit: int = 0
1889:     validation_errors: int = 0
1890:     silent_drops_prevented: int = 0
1891:
1892:
1893: class ExtendedArgRouter(ArgRouter):
1894:     """
1895:     Extended argument router with special route handling.
1896:
1897:     Extends base ArgRouter with:
1898:     - 25+ special route definitions
1899:     - Strict validation (no silent drops)
1900:     - **kwargs awareness for forward compatibility
1901:     - Comprehensive metrics
1902:
1903:     Special Routes (â\211¥25):
1904:     1. _extract_quantitative_claims
```

```
    2. _parse_number
    3. _determine_semantic_role
    4. _compile_pattern_registry
    5. _analyze_temporal_coherence
    6. _validate_evidence_chain
    7. _calculate_confidence_score
    8. _extract_indicators
    9. _parse_temporal_reference
    10. _determine_policy_area
    11. _compile_regex_patterns
    12. _analyze_source_reliability
    13. _validate_numerical_consistency
    14. _calculate_bayesian_update
    15. _extract_entities
    16. _parse_citation
    17. _determine_validation_type
    18. _compile_indicator_patterns
    19. _analyze_coherence_score
    20. _validate_threshold_compliance
    21. _calculate_evidence_weight
    22. _extract_temporal_markers
    23. _parse_budget_allocation
    24. _determine_risk_level
    25. _compile_validation_rules
    26. _analyze_stakeholder_impact
    27. _validate_governance_structure
    28. _calculate_alignment_score
    29. _extract_constraint_declarations
    30. _parse_implementation_timeline
    """

    def __init__(self, class_registry: Mapping[str, type]) -> None:
        """
        Initialize extended router.

        Args:
            class_registry: Mapping of class names to class types
        """
        super().__init__(class_registry)
        self._special_routes = self._build_special_routes()
        self._metrics = RoutingMetrics()
        self._metrics_lock = threading.Lock()

        logger.info(
            "extended_arg_router_initialized",
            special_routes=len(self._special_routes),
            classes=len(class_registry),
        )

    def _build_special_routes(self) -> dict[str, dict[str, Any]]:
        """
        Build special route definitions for commonly-called methods.

        Each route specifies:
        - required_args: List of required parameter names
        - optional_args: List of optional parameter names
```

```
        - accepts_kwargs: Whether method accepts **kwargs
        - description: Human-readable description

        Returns:
            Dict mapping method names to route specs
        """
        routes = {
            "_extract_quantitative_claims": {
                "required_args": ["content"],
                "optional_args": ["context", "thresholds", "patterns"],
                "accepts_kwargs": True,
                "description": "Extract quantitative claims from content",
            },
            "_parse_number": {
                "required_args": ["text"],
                "optional_args": ["locale", "unit_system"],
                "accepts_kwargs": True,
                "description": "Parse numerical value from text",
            },
            "_determine_semantic_role": {
                "required_args": ["text", "context"],
                "optional_args": ["role_taxonomy", "confidence_threshold"],
                "accepts_kwargs": True,
                "description": "Determine semantic role of text element",
            },
            "_compile_pattern_registry": {
                "required_args": ["patterns"],
                "optional_args": ["category", "flags"],
                "accepts_kwargs": False,
                "description": "Compile patterns into regex registry",
            },
            "_analyze_temporal_coherence": {
                "required_args": ["content"],
                "optional_args": ["temporal_patterns", "baseline_date"],
                "accepts_kwargs": True,
                "description": "Analyze temporal coherence of content",
            },
            "_validate_evidence_chain": {
                "required_args": ["claims", "evidence"],
                "optional_args": ["validation_rules", "min_confidence"],
                "accepts_kwargs": True,
                "description": "Validate evidence chain for claims",
            },
            "_calculate_confidence_score": {
                "required_args": ["evidence"],
                "optional_args": ["prior", "weights"],
                "accepts_kwargs": True,
                "description": "Calculate Bayesian confidence score",
            },
            "_extract_indicators": {
                "required_args": ["content"],
                "optional_args": ["indicator_patterns", "extraction_mode"],
                "accepts_kwargs": True,
                "description": "Extract KPI indicators from content",
            },
            "_parse_temporal_reference": {
```

```
                "required_args": ["text"],
                "optional_args": ["reference_date", "format_hints"],
                "accepts_kwargs": True,
                "description": "Parse temporal reference from text",
            },
            "_determine_policy_area": {
                "required_args": ["content"],
                "optional_args": ["taxonomy", "multi_label"],
                "accepts_kwargs": True,
                "description": "Classify content into policy area",
            },
            "_compile_regex_patterns": {
                "required_args": ["pattern_list"],
                "optional_args": ["flags", "validate"],
                "accepts_kwargs": False,
                "description": "Compile list of regex patterns",
            },
            "_analyze_source_reliability": {
                "required_args": ["source"],
                "optional_args": ["source_patterns", "reliability_threshold"],
                "accepts_kwargs": True,
                "description": "Analyze reliability of information source",
            },
            "_validate_numerical_consistency": {
                "required_args": ["numbers"],
                "optional_args": ["tolerance", "consistency_rules"],
                "accepts_kwargs": True,
                "description": "Validate numerical consistency across values",
            },
            "_calculate_bayesian_update": {
                "required_args": ["prior", "likelihood", "evidence"],
                "optional_args": ["normalization"],
                "accepts_kwargs": True,
                "description": "Calculate Bayesian posterior update",
            },
            "_extract_entities": {
                "required_args": ["content"],
                "optional_args": ["entity_types", "confidence_threshold"],
                "accepts_kwargs": True,
                "description": "Extract named entities from content",
            },
            "_parse_citation": {
                "required_args": ["text"],
                "optional_args": ["citation_style", "strict_mode"],
                "accepts_kwargs": True,
                "description": "Parse citation from text",
            },
            "_determine_validation_type": {
                "required_args": ["validation_spec"],
                "optional_args": ["context"],
                "accepts_kwargs": True,
                "description": "Determine type of validation to apply",
            },
            "_compile_indicator_patterns": {
                "required_args": ["indicators"],
                "optional_args": ["category", "weights"],
```

```
                "accepts_kwargs": False,
                "description": "Compile indicator patterns for matching",
            },
            "_analyze_coherence_score": {
                "required_args": ["content"],
                "optional_args": ["coherence_patterns", "scoring_mode"],
                "accepts_kwargs": True,
                "description": "Analyze narrative coherence score",
            },
            "_validate_threshold_compliance": {
                "required_args": ["value", "thresholds"],
                "optional_args": ["strict_mode"],
                "accepts_kwargs": True,
                "description": "Validate value against thresholds",
            },
            "_calculate_evidence_weight": {
                "required_args": ["evidence"],
                "optional_args": ["weighting_scheme", "normalization"],
                "accepts_kwargs": True,
                "description": "Calculate evidence weight for scoring",
            },
            "_extract_temporal_markers": {
                "required_args": ["content"],
                "optional_args": ["temporal_patterns", "extraction_depth"],
                "accepts_kwargs": True,
                "description": "Extract temporal markers from content",
            },
            "_parse_budget_allocation": {
                "required_args": ["text"],
                "optional_args": ["currency", "fiscal_year"],
                "accepts_kwargs": True,
                "description": "Parse budget allocation from text",
            },
            "_determine_risk_level": {
                "required_args": ["indicators"],
                "optional_args": ["risk_thresholds", "aggregation_method"],
                "accepts_kwargs": True,
                "description": "Determine risk level from indicators",
            },
            "_compile_validation_rules": {
                "required_args": ["rules"],
                "optional_args": ["rule_format"],
                "accepts_kwargs": False,
                "description": "Compile validation rules for execution",
            },
            "_analyze_stakeholder_impact": {
                "required_args": ["stakeholders", "policy"],
                "optional_args": ["impact_dimensions", "time_horizon"],
                "accepts_kwargs": True,
                "description": "Analyze stakeholder impact of policy",
            },
            "_validate_governance_structure": {
                "required_args": ["structure"],
                "optional_args": ["governance_standards", "strict_mode"],
                "accepts_kwargs": True,
                "description": "Validate governance structure compliance",
```

```
            },
            "_calculate_alignment_score": {
                "required_args": ["policy_content", "reference_framework"],
                "optional_args": ["alignment_weights", "scoring_method"],
                "accepts_kwargs": True,
                "description": "Calculate alignment score with framework",
            },
            "_extract_constraint_declarations": {
                "required_args": ["content"],
                "optional_args": ["constraint_types", "extraction_mode"],
                "accepts_kwargs": True,
                "description": "Extract constraint declarations from content",
            },
            "_parse_implementation_timeline": {
                "required_args": ["text"],
                "optional_args": ["reference_date", "granularity"],
                "accepts_kwargs": True,
                "description": "Parse implementation timeline from text",
            },
        }

        return routes

    def route(
        self,
        class_name: str,
        method_name: str,
        payload: MutableMapping[str, Any],
    ) -> tuple[tuple[Any, ...], dict[str, Any]]:
        """
        Route method call with special handling and strict validation.

        This override:
        1. Checks for special route definitions
        2. Applies strict validation
        3. Prevents silent parameter drops
        4. Tracks metrics

        Args:
            class_name: Target class name
            method_name: Target method name
            payload: Method parameters

        Returns:
            Tuple of (args, kwargs) for method invocation

        Raises:
            ArgumentValidationError: On validation failure
        """
        with self._metrics_lock:
            self._metrics.total_routes += 1

        # Check for special route
        if method_name in self._special_routes:
            return self._route_special(class_name, method_name, payload)

```

```
2185:         # Use default routing with enhanced validation
2186:         return self._route_default_strict(class_name, method_name, payload)
2187:
2188:     def _route_special(
2189:         self,
2190:         class_name: str,
2191:         method_name: str,
2192:         payload: MutableMapping[str, Any],
2193:     ) -> tuple[tuple[Any, ...], dict[str, Any]]:
2194:         """
2195:         Route using special route definition.
2196:
2197:         Args:
2198:             class_name: Target class name
2199:             method_name: Target method name
2200:             payload: Method parameters
2201:
2202:         Returns:
2203:             Tuple of (args, kwargs)
2204:         """
2205:         with self._metrics_lock:
2206:             self._metrics.special_routes_hit += 1
2207:
2208:         route_spec = self._special_routes[method_name]
2209:         required_args = set(route_spec["required_args"])
2210:         optional_args = set(route_spec["optional_args"])
2211:         accepts_kwargs = route_spec["accepts_kwargs"]
2212:
2213:         provided_keys = set(payload.keys())
2214:
2215:         # Check required arguments
2216:         missing = required_args - provided_keys
2217:         if missing:
2218:             with self._metrics_lock:
2219:                 self._metrics.validation_errors += 1
2220:             logger.error(
2221:                 "special_route_missing_args",
2222:                 class_name=class_name,
2223:                 method=method_name,
2224:                 missing=sorted(missing),
2225:             )
2226:             raise ArgumentValidationError(
2227:                 class_name,
2228:                 method_name,
2229:                 missing=missing,
2230:             )
2231:
2232:         # Check unexpected arguments
2233:         expected = required_args | optional_args
2234:         unexpected = provided_keys - expected
2235:
2236:         if unexpected and not accepts_kwargs:
2237:             # Method doesn't accept **kwargs, so unexpected args are an error
2238:             with self._metrics_lock:
2239:                 self._metrics.validation_errors += 1
2240:                 self._metrics.silent_drops_prevented += 1
```

```
2241:
2242:             logger.error(
2243:                 "special_route_unexpected_args",
2244:                 class_name=class_name,
2245:                 method=method_name,
2246:                 unexpected=sorted(unexpected),
2247:                 accepts_kwargs=accepts_kwargs,
2248:             )
2249:             raise ArgumentValidationError(
2250:                 class_name,
2251:                 method_name,
2252:                 unexpected=unexpected,
2253:             )
2254:
2255:         # Build kwargs (all parameters go to kwargs for special routes)
2256:         kwargs = dict(payload)
2257:
2258:         logger.debug(
2259:             "special_route_applied",
2260:             class_name=class_name,
2261:             method=method_name,
2262:             params_count=len(kwargs),
2263:         )
2264:
2265:         return (), kwargs
2266:
2267:     def _route_default_strict(
2268:         self,
2269:         class_name: str,
2270:         method_name: str,
2271:         payload: MutableMapping[str, Any],
2272:     ) -> tuple[tuple[Any, ...], dict[str, Any]]:
2273:         """
2274:         Route using default strategy with strict validation.
2275:
2276:         This prevents silent parameter drops by failing when:
2277:         - Required arguments are missing
2278:         - Unexpected arguments are provided AND method lacks **kwargs
2279:
2280:         Args:
2281:             class_name: Target class name
2282:             method_name: Target method name
2283:             payload: Method parameters
2284:
2285:         Returns:
2286:             Tuple of (args, kwargs)
2287:         """
2288:         with self._metrics_lock:
2289:             self._metrics.default_routes_hit += 1
2290:
2291:         # Use base implementation for inspection
2292:         spec = self.describe(class_name, method_name)
2293:
2294:         # Strict validation: if unexpected args and no **kwargs, fail
2295:         provided_keys = set(payload.keys())
2296:         accepted = set(spec.accepted_arguments)
```

```
        unexpected = provided_keys - accepted

        if unexpected and not spec.has_var_keyword:
            # Method doesn't accept **kwargs - unexpected args are errors
            with self._metrics_lock:
                self._metrics.validation_errors += 1
                self._metrics.silent_drops_prevented += 1

            logger.error(
                "default_route_unexpected_args_strict",
                class_name=class_name,
                method=method_name,
                unexpected=sorted(unexpected),
                has_var_keyword=spec.has_var_keyword,
            )
            raise ArgumentValidationError(
                class_name,
                method_name,
                unexpected=unexpected,
            )

        # Delegate to base implementation
        try:
            result = super().route(class_name, method_name, payload)
            logger.debug(
                "default_route_applied",
                class_name=class_name,
                method=method_name,
            )
            return result
        except ArgumentValidationError:
            with self._metrics_lock:
                self._metrics.validation_errors += 1
            raise

    def get_special_route_coverage(self) -> int:
        """
        Get count of special routes defined.

        Returns:
            Number of special routes (target: â\211¥25)
        """
        return len(self._special_routes)

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

        Returns:
            Dict with routing statistics
        """
        total = self._metrics.total_routes or 1  # Avoid division by zero

        return {
            "total_routes": self._metrics.total_routes,
            "special_routes_hit": self._metrics.special_routes_hit,
```

```
            "special_routes_coverage": len(self._special_routes),
            "default_routes_hit": self._metrics.default_routes_hit,
            "validation_errors": self._metrics.validation_errors,
            "silent_drops_prevented": self._metrics.silent_drops_prevented,
            "special_route_hit_rate": self._metrics.special_routes_hit / total,
            "error_rate": self._metrics.validation_errors / total,
        }

    def list_special_routes(self) -> list[dict[str, Any]]:
        """
        List all special routes with their specifications.

        Returns:
            List of route specifications
        """
        routes = []
        for method_name, spec in sorted(self._special_routes.items()):
            routes.append({
                "method_name": method_name,
                "required_args": spec["required_args"],
                "optional_args": spec["optional_args"],
                "accepts_kwargs": spec["accepts_kwargs"],
                "description": spec["description"],
            })
        return routes


================================================================================
FILE: src/farfan_pipeline/core/orchestrator/base_executor_with_contract.py
================================================================================

from __future__ import annotations

import json
from abc import ABC, abstractmethod
from typing import TYPE_CHECKING, Any

from jsonschema import Draft7Validator

from farfan_pipeline.config.paths import PROJECT_ROOT
from farfan_pipeline.core.orchestrator.evidence_assembler import EvidenceAssembler
from farfan_pipeline.core.orchestrator.evidence_registry import get_global_registry
from farfan_pipeline.core.orchestrator.evidence_validator import EvidenceValidator

if TYPE_CHECKING:
    from farfan_pipeline.core.orchestrator.core import MethodExecutor
    from farfan_pipeline.core.types import PreprocessedDocument
else:  # pragma: no cover - runtime avoids import to break cycles
    MethodExecutor = Any
    PreprocessedDocument = Any


class BaseExecutorWithContract(ABC):
    """Contract-driven executor that routes all calls through MethodExecutor.

```

```
    Supports both v2 and v3 contract formats:
    - v2: Legacy format with method_inputs, assembly_rules, validation_rules at top level
    - v3: New format with identity, executor_binding, method_binding, question_context,
          evidence_assembly, output_contract, validation_rules, etc.

    Contract version is auto-detected based on file name (.v3.json vs .json) and structure.
    """

    _contract_cache: dict[str, dict[str, Any]] = {}
    _schema_validators: dict[str, Draft7Validator] = {}
    _factory_contracts_verified: bool = False
    _factory_verification_errors: list[str] = []

    def __init__(
        self,
        method_executor: MethodExecutor,
        signal_registry: Any,
        config: Any,
        questionnaire_provider: Any,
        calibration_orchestrator: Any | None = None,
        enriched_packs: dict[str, Any] | None = None,
        validation_orchestrator: Any | None = None,
    ) -> None:
        try:
            from farfan_pipeline.core.orchestrator.core import (
                MethodExecutor as _MethodExecutor,
            )
        except Exception as exc:  # pragma: no cover - defensive guard
            raise RuntimeError(
                "Failed to import MethodExecutor for BaseExecutorWithContract invariants. "
                "Ensure farfan_core.core.orchestrator.core is importable before constructing contract executors."
            ) from exc
        if not isinstance(method_executor, _MethodExecutor):
            raise RuntimeError(
                "A valid MethodExecutor instance is required for contract executors."
            )
        self.method_executor = method_executor
        self.signal_registry = signal_registry
        self.config = config
        self.questionnaire_provider = questionnaire_provider
        self.calibration_orchestrator = calibration_orchestrator
        # JOBFRONT 3: Support for enriched signal packs (intelligence layer)
        self.enriched_packs = enriched_packs or {}
        self._use_enriched_signals = len(self.enriched_packs) > 0
        # VALIDATION ORCHESTRATOR: Comprehensive validation tracking
        self.validation_orchestrator = validation_orchestrator
        self._use_validation_orchestrator = validation_orchestrator is not None

    @classmethod
    @abstractmethod
    def get_base_slot(cls) -> str:
        raise NotImplementedError

    @classmethod
    def verify_all_base_contracts(
        cls, class_registry: dict[str, type[object]] | None = None
```

```
    ) -> dict[str, Any]:
        """Verify all 30 base executor contracts at factory initialization time.

        This method loads and validates all contracts for D1-Q1 through D6-Q5, checking:
        - Contract files exist and are valid JSON
        - Required fields are present (method_inputs/method_binding, assembly_rules,
          validation_rules, expected_elements)
        - JSON schema compliance (v2 or v3)
        - All referenced method classes exist in the class registry

        Args:
            class_registry: Optional class registry to verify method class existence.
                            If None, will attempt to import and build one.

        Returns:
            dict with keys:
                - passed: bool indicating if all contracts are valid
                - total_contracts: int count of contracts checked
                - errors: list of error messages for failed contracts
                - warnings: list of warning messages
                - verified_contracts: list of base_slot identifiers that passed

        Raises:
            RuntimeError: If verification fails with strict=True
        """
        if cls._factory_contracts_verified:
            return {
                "passed": len(cls._factory_verification_errors) == 0,
                "total_contracts": 30,
                "errors": cls._factory_verification_errors,
                "warnings": [],
                "verified_contracts": list(cls._contract_cache.keys()),
            }

        base_slots = [
            f"D{d}-Q{q}" for d in range(1, 7) for q in range(1, 6)
        ]

        if class_registry is None:
            try:
                from farfan_pipeline.core.orchestrator.class_registry import (
                    build_class_registry,
                )
                class_registry = build_class_registry()
            except Exception as exc:
                cls._factory_verification_errors.append(
                    f"Failed to build class registry for verification: {exc}"
                )

        errors: list[str] = []
        warnings: list[str] = []
        verified_contracts: list[str] = []

        for base_slot in base_slots:
            try:
                result = cls._verify_single_contract(base_slot, class_registry)
```

```
                if result["passed"]:
                    verified_contracts.append(base_slot)
                else:
                    errors.extend(
                        f"[{base_slot}] {err}" for err in result["errors"]
                    )
                warnings.extend(
                    f"[{base_slot}] {warn}" for warn in result.get("warnings", [])
                )
            except Exception as exc:
                errors.append(f"[{base_slot}] Unexpected error during verification: {exc}")

        cls._factory_contracts_verified = True
        cls._factory_verification_errors = errors

        return {
            "passed": len(errors) == 0,
            "total_contracts": len(base_slots),
            "errors": errors,
            "warnings": warnings,
            "verified_contracts": verified_contracts,
        }

    @classmethod
    def _verify_single_contract(
        cls, base_slot: str, class_registry: dict[str, type[object]] | None = None
    ) -> dict[str, Any]:
        """Verify a single contract for completeness and validity.

        Args:
            base_slot: Base slot identifier (e.g., "D1-Q1")
            class_registry: Optional class registry for method class verification

        Returns:
            dict with keys:
                - passed: bool
                - errors: list of error messages
                - warnings: list of warning messages
                - contract_version: detected version (v2/v3)
                - contract_path: path to contract file
        """
        errors: list[str] = []
        warnings: list[str] = []

        dimension = int(base_slot[1])
        question = int(base_slot[4])
        q_number = (dimension - 1) * 5 + question
        q_id = f"Q{q_number:03d}"

        v3_path = (
            PROJECT_ROOT / "config" / "executor_contracts" / f"{base_slot}.v3.json"
        )
        v2_path = PROJECT_ROOT / "config" / "executor_contracts" / f"{base_slot}.json"
        v3_specialized_path = (
            PROJECT_ROOT / "config" / "executor_contracts" / "specialized" / f"{q_id}.v3.json"
        )
```

```
2577:         v2_specialized_path = (
2578:             PROJECT_ROOT / "config" / "executor_contracts" / "specialized" / f"{q_id}.json"
2579:         )
2580:
2581:         contract_path = None
2582:         if v3_path.exists():
2583:             contract_path = v3_path
2584:             expected_version = "v3"
2585:         elif v2_path.exists():
2586:             contract_path = v2_path
2587:             expected_version = "v2"
2588:         elif v3_specialized_path.exists():
2589:             contract_path = v3_specialized_path
2590:             expected_version = "v3"
2591:         elif v2_specialized_path.exists():
2592:             contract_path = v2_specialized_path
2593:             expected_version = "v2"
2594:         else:
2595:             errors.append(
2596:                 f"Contract file not found. Tried: {v3_path}, {v2_path}, {v3_specialized_path}, {v2_specialized_path}"
2597:             )
2598:             return {
2599:                 "passed": False,
2600:                 "errors": errors,
2601:                 "warnings": warnings,
2602:                 "contract_version": None,
2603:                 "contract_path": None,
2604:             }
2605:
2606:         try:
2607:             contract = json.loads(contract_path.read_text(encoding="utf-8"))
2608:         except json.JSONDecodeError as exc:
2609:             errors.append(f"Invalid JSON in contract file: {exc}")
2610:             return {
2611:                 "passed": False,
2612:                 "errors": errors,
2613:                 "warnings": warnings,
2614:                 "contract_version": expected_version,
2615:                 "contract_path": str(contract_path),
2616:             }
2617:         except Exception as exc:
2618:             errors.append(f"Failed to read contract file: {exc}")
2619:             return {
2620:                 "passed": False,
2621:                 "errors": errors,
2622:                 "warnings": warnings,
2623:                 "contract_version": expected_version,
2624:                 "contract_path": str(contract_path),
2625:             }
2626:
2627:         detected_version = cls._detect_contract_version(contract)
2628:         if detected_version != expected_version:
2629:             warnings.append(
2630:                 f"Contract structure is {detected_version} but file naming suggests {expected_version}"
2631:             )
2632:
```

```
2633:         try:
2634:             validator = cls._get_schema_validator(detected_version)
2635:             schema_errors = sorted(validator.iter_errors(contract), key=lambda e: e.path)
2636:             if schema_errors:
2637:                 errors.extend(
2638:                     f"Schema validation error: {err.message} at {'.'.join(str(p) for p in err.path)}"
2639:                     for err in schema_errors[:10]
2640:                 )
2641:         except FileNotFoundError as exc:
2642:             warnings.append(f"Schema file not found: {exc}. Skipping schema validation.")
2643:         except Exception as exc:
2644:             warnings.append(f"Schema validation error: {exc}")
2645:
2646:         if detected_version == "v3":
2647:             v3_errors = cls._verify_v3_contract_fields(contract, base_slot, class_registry)
2648:             errors.extend(v3_errors)
2649:         else:
2650:             v2_errors = cls._verify_v2_contract_fields(contract, base_slot, class_registry)
2651:             errors.extend(v2_errors)
2652:
2653:         return {
2654:             "passed": len(errors) == 0,
2655:             "errors": errors,
2656:             "warnings": warnings,
2657:             "contract_version": detected_version,
2658:             "contract_path": str(contract_path),
2659:         }
2660:
2661:     @classmethod
2662:     def _verify_v2_contract_fields(
2663:         cls,
2664:         contract: dict[str, Any],
2665:         base_slot: str,
2666:         class_registry: dict[str, type[object]] | None = None,
2667:     ) -> list[str]:
2668:         """Verify required fields for v2 contract format.
2669:
2670:         Args:
2671:             contract: Parsed contract dict
2672:             base_slot: Base slot identifier
2673:             class_registry: Optional class registry for method verification
2674:
2675:         Returns:
2676:             List of error messages (empty if all checks pass)
2677:         """
2678:         errors: list[str] = []
2679:
2680:         if "method_inputs" not in contract:
2681:             errors.append("Missing required field: method_inputs")
2682:         elif not isinstance(contract["method_inputs"], list):
2683:             errors.append("method_inputs must be a list")
2684:         else:
2685:             method_inputs = contract["method_inputs"]
2686:             if not method_inputs:
2687:                 errors.append("method_inputs is empty")
2688:             else:
```

```
                for idx, method_spec in enumerate(method_inputs):
                    if not isinstance(method_spec, dict):
                        errors.append(f"method_inputs[{idx}] is not a dict")
                        continue
                    if "class" not in method_spec:
                        errors.append(f"method_inputs[{idx}] missing 'class' field")
                    if "method" not in method_spec:
                        errors.append(f"method_inputs[{idx}] missing 'method' field")

                    if class_registry is not None and "class" in method_spec:
                        class_name = method_spec["class"]
                        if class_name not in class_registry:
                            errors.append(
                                f"method_inputs[{idx}]: class '{class_name}' not found in class registry"
                            )

        if "assembly_rules" not in contract:
            errors.append("Missing required field: assembly_rules")
        elif not isinstance(contract["assembly_rules"], list):
            errors.append("assembly_rules must be a list")

        if "validation_rules" not in contract:
            errors.append("Missing required field: validation_rules")

        return errors

    @classmethod
    def _verify_v3_contract_fields(
        cls,
        contract: dict[str, Any],
        base_slot: str,
        class_registry: dict[str, type[object]] | None = None,
    ) -> list[str]:
        """Verify required fields for v3 contract format.

        Args:
            contract: Parsed contract dict
            base_slot: Base slot identifier
            class_registry: Optional class registry for method verification

        Returns:
            List of error messages (empty if all checks pass)
        """
        errors: list[str] = []

        if "identity" not in contract:
            errors.append("Missing required field: identity")
        else:
            identity = contract["identity"]
            if "base_slot" not in identity:
                errors.append("identity missing 'base_slot' field")
            elif identity["base_slot"] != base_slot:
                errors.append(
                    f"identity.base_slot mismatch: expected {base_slot}, got {identity['base_slot']}"
                )

```

```
        if "method_binding" not in contract:
            errors.append("Missing required field: method_binding")
        else:
            method_binding = contract["method_binding"]
            orchestration_mode = method_binding.get("orchestration_mode", "single_method")

            if orchestration_mode == "multi_method_pipeline":
                if "methods" not in method_binding:
                    errors.append("method_binding missing 'methods' array for multi_method_pipeline mode")
                elif not isinstance(method_binding["methods"], list):
                    errors.append("method_binding.methods must be a list")
                else:
                    methods = method_binding["methods"]
                    if not methods:
                        errors.append("method_binding.methods is empty")
                    else:
                        for idx, method_spec in enumerate(methods):
                            if not isinstance(method_spec, dict):
                                errors.append(f"methods[{idx}] is not a dict")
                                continue
                            if "class_name" not in method_spec:
                                errors.append(f"methods[{idx}] missing 'class_name' field")
                            if "method_name" not in method_spec:
                                errors.append(f"methods[{idx}] missing 'method_name' field")

                            if class_registry is not None and "class_name" in method_spec:
                                class_name = method_spec["class_name"]
                                if class_name not in class_registry:
                                    errors.append(
                                        f"methods[{idx}]: class '{class_name}' not found in class registry"
                                    )
            elif "class_name" not in method_binding and "primary_method" not in method_binding:
                errors.append(
                    "method_binding missing 'class_name' or 'primary_method' for single_method mode"
                )
            else:
                class_name = method_binding.get("class_name")
                if not class_name and "primary_method" in method_binding:
                    class_name = method_binding["primary_method"].get("class_name")

                if class_name and class_registry is not None:
                    if class_name not in class_registry:
                        errors.append(
                            f"method_binding: class '{class_name}' not found in class registry"
                        )

        if "evidence_assembly" not in contract:
            errors.append("Missing required field: evidence_assembly")
        else:
            evidence_assembly = contract["evidence_assembly"]
            if "assembly_rules" not in evidence_assembly:
                errors.append("evidence_assembly missing 'assembly_rules' field")
            elif not isinstance(evidence_assembly["assembly_rules"], list):
                errors.append("evidence_assembly.assembly_rules must be a list")

        if "validation_rules" not in contract:
```

```
            errors.append("Missing required field: validation_rules")

        if "question_context" not in contract:
            errors.append("Missing required field: question_context")
        else:
            question_context = contract["question_context"]
            if "expected_elements" not in question_context:
                errors.append("question_context missing 'expected_elements' field")

        if "error_handling" not in contract:
            errors.append("Missing required field: error_handling")

        return errors

    @classmethod
    def _get_schema_validator(cls, version: str = "v2") -> Draft7Validator:
        """Get schema validator for the specified contract version.

        Args:
            version: Contract version ("v2" or "v3")

        Returns:
            Draft7Validator for the specified version
        """
        if version not in cls._schema_validators:
            if version == "v3":
                schema_path = (
                    PROJECT_ROOT
                    / "config"
                    / "schemas"
                    / "executor_contract.v3.schema.json"
                )
            else:
                schema_path = PROJECT_ROOT / "config" / "executor_contract.schema.json"

            if not schema_path.exists():
                raise FileNotFoundError(f"Contract schema not found: {schema_path}")
            schema = json.loads(schema_path.read_text(encoding="utf-8"))
            cls._schema_validators[version] = Draft7Validator(schema)
        return cls._schema_validators[version]

    @classmethod
    def _detect_contract_version(cls, contract: dict[str, Any]) -> str:
        """Detect contract version from structure.

        v3 contracts have: identity, executor_binding, method_binding, question_context
        v2 contracts have: method_inputs, assembly_rules at top level

        Returns:
            "v3" or "v2"
        """
        v3_indicators = [
            "identity",
            "executor_binding",
            "method_binding",
            "question_context",
```

```
2857:         ]
2858:         if all(key in contract for key in v3_indicators):
2859:             return "v3"
2860:         return "v2"
2861:
2862:     @classmethod
2863:     def _load_contract(cls) -> dict[str, Any]:
2864:         base_slot = cls.get_base_slot()
2865:         if base_slot in cls._contract_cache:
2866:             return cls._contract_cache[base_slot]
2867:
2868:         dimension = int(base_slot[1])
2869:         question = int(base_slot[4])
2870:         q_number = (dimension - 1) * 5 + question
2871:         q_id = f"Q{q_number:03d}"
2872:
2873:         v3_path = (
2874:             PROJECT_ROOT / "config" / "executor_contracts" / f"{base_slot}.v3.json"
2875:         )
2876:         v2_path = PROJECT_ROOT / "config" / "executor_contracts" / f"{base_slot}.json"
2877:         v3_specialized_path = (
2878:             PROJECT_ROOT / "config" / "executor_contracts" / "specialized" / f"{q_id}.v3.json"
2879:         )
2880:         v2_specialized_path = (
2881:             PROJECT_ROOT / "config" / "executor_contracts" / "specialized" / f"{q_id}.json"
2882:         )
2883:
2884:         if v3_path.exists():
2885:             contract_path = v3_path
2886:             expected_version = "v3"
2887:         elif v2_path.exists():
2888:             contract_path = v2_path
2889:             expected_version = "v2"
2890:         elif v3_specialized_path.exists():
2891:             contract_path = v3_specialized_path
2892:             expected_version = "v3"
2893:         elif v2_specialized_path.exists():
2894:             contract_path = v2_specialized_path
2895:             expected_version = "v2"
2896:         else:
2897:             raise FileNotFoundError(
2898:                 f"Contract not found for {base_slot}. "
2899:                 f"Tried: {v3_path}, {v2_path}, {v3_specialized_path}, {v2_specialized_path}"
2900:             )
2901:
2902:         contract = json.loads(contract_path.read_text(encoding="utf-8"))
2903:
2904:         # Detect actual version from structure
2905:         detected_version = cls._detect_contract_version(contract)
2906:         if detected_version != expected_version:
2907:             import logging
2908:
2909:             logging.warning(
2910:                 f"Contract {contract_path.name} has structure of {detected_version} "
2911:                 f"but file naming suggests {expected_version}"
2912:             )
```

```
2913:
2914:         # Validate with appropriate schema
2915:         validator = cls._get_schema_validator(detected_version)
2916:         errors = sorted(validator.iter_errors(contract), key=lambda e: e.path)
2917:         if errors:
2918:             messages = "; ".join(err.message for err in errors)
2919:             raise ValueError(
2920:                 f"Contract validation failed for {base_slot} ({detected_version}): {messages}"
2921:             )
2922:
2923:         # Tag contract with version for later use
2924:         contract["_contract_version"] = detected_version
2925:
2926:         contract_version = contract.get("contract_version")
2927:         if contract_version and not str(contract_version).startswith("2"):
2928:             raise ValueError(
2929:                 f"Unsupported contract_version {contract_version} for {base_slot}; expected v2.x"
2930:             )
2931:
2932:         identity_base_slot = contract.get("identity", {}).get("base_slot")
2933:         if identity_base_slot and identity_base_slot != base_slot:
2934:             raise ValueError(
2935:                 f"Contract base_slot mismatch: expected {base_slot}, found {identity_base_slot}"
2936:             )
2937:
2938:         cls._contract_cache[base_slot] = contract
2939:         return contract
2940:
2941:     def _validate_signal_requirements(
2942:         self,
2943:         signal_pack: Any,
2944:         signal_requirements: dict[str, Any],
2945:         base_slot: str,
2946:     ) -> None:
2947:         """Validate that signal requirements from contract are met.
2948:
2949:         Args:
2950:             signal_pack: Signal pack retrieved from registry (may be None)
2951:             signal_requirements: signal_requirements section from contract
2952:             base_slot: Base slot identifier for error messages
2953:
2954:         Raises:
2955:             RuntimeError: If mandatory signal requirements are not met
2956:         """
2957:         mandatory_signals = signal_requirements.get("mandatory_signals", [])
2958:         minimum_threshold = signal_requirements.get("minimum_signal_threshold", 0.0)
2959:
2960:         # Check if mandatory signals are required but no signal pack available
2961:         if mandatory_signals and signal_pack is None:
2962:             raise RuntimeError(
2963:                 f"Contract {base_slot} requires mandatory signals {mandatory_signals}, "
2964:                 "but no signal pack was retrieved from registry. "
2965:                 "Ensure signal registry is properly configured and policy_area_id is valid."
2966:             )
2967:
2968:         # If signal pack exists, validate signal strength
```

```
2969:         if signal_pack is not None and minimum_threshold > 0:
2970:             # Check if signal pack has strength attribute
2971:             if hasattr(signal_pack, "strength") or (
2972:                 isinstance(signal_pack, dict) and "strength" in signal_pack
2973:             ):
2974:                 strength = (
2975:                     signal_pack.strength
2976:                     if hasattr(signal_pack, "strength")
2977:                     else signal_pack["strength"]
2978:                 )
2979:                 if strength < minimum_threshold:
2980:                     raise RuntimeError(
2981:                         f"Contract {base_slot} requires minimum signal threshold {minimum_threshold}, "
2982:                         f"but signal pack has strength {strength}. "
2983:                         "Signal quality is insufficient for execution."
2984:                     )
2985:
2986:     @staticmethod
2987:     def _set_nested_value(
2988:         target_dict: dict[str, Any], key_path: str, value: Any
2989:     ) -> None:
2990:         """Set a value in a nested dict using dot-notation key path.
2991:
2992:         Args:
2993:             target_dict: The dictionary to modify
2994:             key_path: Dot-separated path (e.g., "text_mining.critical_links")
2995:             value: The value to set
2996:
2997:         Example:
2998:             _set_nested_value(d, "a.b.c", 123) â\206\222 d["a"]["b"]["c"] = 123
2999:         """
3000:         keys = key_path.split(".")
3001:         current = target_dict
3002:
3003:         # Navigate to the parent of the final key, creating dicts as needed
3004:         for key in keys[:-1]:
3005:             if key not in current:
3006:                 current[key] = {}
3007:             elif not isinstance(current[key], dict):
3008:                 # Key exists but is not a dict, cannot nest further
3009:                 raise ValueError(
3010:                     f"Cannot set nested value at '{key_path}': "
3011:                     f"intermediate key '{key}' exists but is not a dict"
3012:                 )
3013:             current = current[key]
3014:
3015:         # Set the final key
3016:         current[keys[-1]] = value
3017:
3018:     def _check_failure_contract(
3019:         self, evidence: dict[str, Any], error_handling: dict[str, Any]
3020:     ) -> None:
3021:         failure_contract = error_handling.get("failure_contract", {})
3022:         abort_conditions = failure_contract.get("abort_if", [])
3023:         if not abort_conditions:
3024:             return
```

```
3025:
3026:         emit_code = failure_contract.get("emit_code", "GENERIC_ABORT")
3027:
3028:         for condition in abort_conditions:
3029:             # Example condition check. This could be made more sophisticated.
3030:             if condition == "missing_required_element" and evidence.get(
3031:                 "validation", {}
3032:             ).get("errors"):
3033:                 # This logic assumes errors from the validator imply a missing required element,
3034:                 # which is true with our new validator.
3035:                 raise ValueError(
3036:                     f"Execution aborted by failure contract due to '{condition}'. Emit code: {emit_code}"
3037:                 )
3038:             if condition == "incomplete_text" and not evidence.get("metadata", {}).get(
3039:                 "text_complete", True
3040:             ):
3041:                 raise ValueError(
3042:                     f"Execution aborted by failure contract due to '{condition}'. Emit code: {emit_code}"
3043:                 )
3044:
3045:     def execute(
3046:         self,
3047:         document: PreprocessedDocument,
3048:         method_executor: MethodExecutor,
3049:         *,
3050:         question_context: dict[str, Any],
3051:     ) -> dict[str, Any]:
3052:         if method_executor is not self.method_executor:
3053:             raise RuntimeError(
3054:                 "Mismatched MethodExecutor instance for contract executor"
3055:             )
3056:
3057:         base_slot = self.get_base_slot()
3058:         if question_context.get("base_slot") != base_slot:
3059:             raise ValueError(
3060:                 f"Question base_slot {question_context.get('base_slot')} does not match executor {base_slot}"
3061:             )
3062:
3063:         contract = self._load_contract()
3064:         contract_version = contract.get("_contract_version", "v2")
3065:
3066:         if contract_version == "v3":
3067:             return self._execute_v3(document, question_context, contract)
3068:         else:
3069:             return self._execute_v2(document, question_context, contract)
3070:
3071:     def _execute_v2(
3072:         self,
3073:         document: PreprocessedDocument,
3074:         question_context: dict[str, Any],
3075:         contract: dict[str, Any],
3076:     ) -> dict[str, Any]:
3077:         """Execute using v2 contract format (legacy)."""
3078:         base_slot = self.get_base_slot()
3079:         question_id = question_context.get("question_id")
3080:         question_global = question_context.get("question_global")
```

```
3081:         policy_area_id = question_context.get("policy_area_id")
3082:         identity = question_context.get("identity", {})
3083:         patterns = question_context.get("patterns", [])
3084:         expected_elements = question_context.get("expected_elements", [])
3085:
3086:         # JOBFRONT 3: Use enriched signal packs if available
3087:         signal_pack = None
3088:         enriched_pack = None
3089:         applicable_patterns = patterns  # Default to contract patterns
3090:         document_context = {}
3091:
3092:         if self._use_enriched_signals and policy_area_id in self.enriched_packs:
3093:             # Use enriched intelligence layer
3094:             enriched_pack = self.enriched_packs[policy_area_id]
3095:             signal_pack = enriched_pack.base_pack  # Maintain compatibility
3096:
3097:             # Create document context from available metadata
3098:             from farfan_pipeline.core.orchestrator.signal_intelligence_layer import (
3099:                 create_document_context,
3100:             )
3101:
3102:             doc_metadata = getattr(document, "metadata", {})
3103:             document_context = create_document_context(
3104:                 section=doc_metadata.get("section"),
3105:                 chapter=doc_metadata.get("chapter"),
3106:                 page=doc_metadata.get("page"),
3107:                 policy_area=policy_area_id
3108:             )
3109:
3110:             # Get context-filtered patterns (REFACTORING #6: context scoping)
3111:             applicable_patterns = enriched_pack.get_patterns_for_context(document_context)
3112:
3113:             # Expand patterns semantically (REFACTORING #2: semantic expansion)
3114:             if applicable_patterns and isinstance(applicable_patterns[0], dict):
3115:                 pattern_strings = [p.get('pattern', p) if isinstance(p, dict) else p for p in applicable_patterns]
3116:             else:
3117:                 pattern_strings = applicable_patterns
3118:
3119:             expanded_patterns = enriched_pack.expand_patterns(pattern_strings)
3120:             applicable_patterns = expanded_patterns
3121:
3122:         elif self.signal_registry is not None and hasattr(self.signal_registry, "get") and policy_area_id:
3123:             # Fallback to legacy signal registry
3124:             signal_pack = self.signal_registry.get(policy_area_id)
3125:
3126:         common_kwargs: dict[str, Any] = {
3127:             "document": document,
3128:             "base_slot": base_slot,
3129:             "raw_text": getattr(document, "raw_text", None),
3130:             "text": getattr(document, "raw_text", None),
3131:             "question_id": question_id,
3132:             "question_global": question_global,
3133:             "policy_area_id": policy_area_id,
3134:             "dimension_id": identity.get("dimension_id"),
3135:             "cluster_id": identity.get("cluster_id"),
3136:             "signal_pack": signal_pack,
```

```
            "enriched_pack": enriched_pack,  # NEW: Pass enriched pack
            "document_context": document_context,  # NEW: Pass document context
            "question_patterns": applicable_patterns,  # Use filtered/expanded patterns
            "expected_elements": expected_elements,
        }

        method_outputs: dict[str, Any] = {}
        method_inputs = contract.get("method_inputs", [])
        indexed = list(enumerate(method_inputs))
        sorted_inputs = sorted(
            indexed, key=lambda pair: (pair[1].get("priority", 2), pair[0])
        )
        for _, entry in sorted_inputs:
            class_name = entry["class"]
            method_name = entry["method"]
            provides = entry.get("provides", [])
            extra_args = entry.get("args", {})

            payload = {**common_kwargs, **extra_args}

            result = self.method_executor.execute(
                class_name=class_name,
                method_name=method_name,
                **payload,
            )

            if "signal_pack" in payload and payload["signal_pack"] is not None:
                if "_signal_usage" not in method_outputs:
                    method_outputs["_signal_usage"] = []
                method_outputs["_signal_usage"].append(
                    {
                        "method": f"{class_name}.{method_name}",
                        "policy_area": payload["signal_pack"].policy_area,
                        "version": payload["signal_pack"].version,
                    }
                )

            if isinstance(provides, str):
                method_outputs[provides] = result
            else:
                for key in provides:
                    method_outputs[key] = result

        assembly_rules = contract.get("assembly_rules", [])
        assembled = EvidenceAssembler.assemble(method_outputs, assembly_rules)
        evidence = assembled["evidence"]
        trace = assembled["trace"]

        # JOBFRONT 3: Extract structured evidence if enriched pack available
        completeness = 1.0
        missing_elements = []
        patterns_used = []

        if enriched_pack is not None and expected_elements:
            # Build signal node for evidence extraction
            signal_node = {
```

```
                "id": question_id,
                "expected_elements": expected_elements,
                "patterns": applicable_patterns,
                "validations": contract.get("validation_rules", [])
            }

            # Extract structured evidence (REFACTORING #5: evidence structure)
            evidence_result = enriched_pack.extract_evidence(
                text=getattr(document, "raw_text", ""),
                signal_node=signal_node,
                document_context=document_context
            )

            # Merge structured evidence into result
            for element_type, matches in evidence_result.evidence.items():
                if element_type not in evidence:
                    evidence[element_type] = matches

            completeness = evidence_result.completeness
            missing_elements = evidence_result.missing_required

            # Track patterns used (for confidence calculation)
            if isinstance(applicable_patterns, list):
                patterns_used = [p.get('id', p) if isinstance(p, dict) else p
                                 for p in applicable_patterns[:10]]  # Top 10

        validation_rules = contract.get("validation_rules", [])
        na_policy = contract.get("na_policy", "abort")
        validation_rules_object = {"rules": validation_rules, "na_policy": na_policy}
        validation = EvidenceValidator.validate(evidence, validation_rules_object)

        error_handling = contract.get("error_handling", {})

        # JOBFRONT 3: Add contract validation if enriched pack available
        contract_validation = None
        if enriched_pack is not None:
            # Build signal node for contract validation
            signal_node_for_validation = {
                "id": question_id,
                "failure_contract": error_handling.get("failure_contract", {}),
                "validations": validation_rules,
                "expected_elements": expected_elements
            }

            # Validate with contracts (REFACTORING #4: contract validation)
            from farfan_pipeline.core.orchestrator.signal_contract_validator import (
                validate_result_with_orchestrator,
            )

            contract_validation = validate_result_with_orchestrator(
                result=evidence,
                signal_node=signal_node_for_validation,
                orchestrator=self.validation_orchestrator if self._use_validation_orchestrator else None,
                auto_register=self._use_validation_orchestrator
            )

```

```
            # Merge contract validation into standard validation
            if not contract_validation.passed:
                validation["status"] = "failed"
                validation["errors"] = validation.get("errors", [])
                validation["errors"].append({
                    "error_code": contract_validation.error_code,
                    "condition_violated": contract_validation.condition_violated,
                    "remediation": contract_validation.remediation,
                    "failures_detailed": [
                        {
                            "type": f.failure_type,
                            "field": f.field_name,
                            "message": f.message,
                            "severity": f.severity,
                            "remediation": f.remediation
                        }
                        for f in contract_validation.failures_detailed[:5]
                    ]
                })
                validation["contract_failed"] = True
                validation["contract_validation_details"] = {
                    "error_code": contract_validation.error_code,
                    "diagnostics": contract_validation.diagnostics,
                    "total_failures": len(contract_validation.failures_detailed)
                }
        elif self._use_validation_orchestrator:
            # Even without enriched pack, use validation orchestrator with basic validation
            from farfan_pipeline.core.orchestrator.signal_contract_validator import (
                validate_result_with_orchestrator,
            )

            signal_node_for_validation = {
                "id": question_id,
                "failure_contract": error_handling.get("failure_contract", {}),
                "validations": {"rules": validation_rules},
                "expected_elements": expected_elements
            }

            contract_validation = validate_result_with_orchestrator(
                result=evidence,
                signal_node=signal_node_for_validation,
                orchestrator=self.validation_orchestrator,
                auto_register=True
            )
        if error_handling:
            evidence_with_validation = {**evidence, "validation": validation}
            self._check_failure_contract(evidence_with_validation, error_handling)

        human_answer_template = contract.get("human_answer_template", "")
        human_answer = ""
        if human_answer_template:
            try:
                human_answer = human_answer_template.format(**evidence)
            except KeyError as e:
                human_answer = f"Error formatting human answer: Missing key {e}. Template: '{human_answer_template}'"
                import logging
```

```
3305:
3306:                 logging.warning(human_answer)
3307:
3308:         result = {
3309:             "base_slot": base_slot,
3310:             "question_id": question_id,
3311:             "question_global": question_global,
3312:             "policy_area_id": policy_area_id,
3313:             "dimension_id": identity.get("dimension_id"),
3314:             "cluster_id": identity.get("cluster_id"),
3315:             "evidence": evidence,
3316:             "validation": validation,
3317:             "trace": trace,
3318:             "human_answer": human_answer,
3319:             # JOBFRONT 3: Add intelligence layer metadata
3320:             "completeness": completeness,
3321:             "missing_elements": missing_elements,
3322:             "patterns_used": patterns_used,
3323:             "enriched_signals_enabled": enriched_pack is not None,
3324:             # VALIDATION ORCHESTRATOR: Add validation tracking metadata
3325:             "contract_validation": {
3326:                 "enabled": contract_validation is not None,
3327:                 "passed": contract_validation.passed if contract_validation else None,
3328:                 "error_code": contract_validation.error_code if contract_validation else None,
3329:                 "failure_count": len(contract_validation.failures_detailed) if contract_validation else 0,
3330:                 "orchestrator_registered": self._use_validation_orchestrator
3331:             }
3332:         }
3333:
3334:         return result
3335:
3336:     def _execute_v3(
3337:         self,
3338:         document: PreprocessedDocument,
3339:         question_context_external: dict[str, Any],
3340:         contract: dict[str, Any],
3341:     ) -> dict[str, Any]:
3342:         """Execute using v3 contract format.
3343:
3344:         In v3, contract contains all context, so we use contract['question_context']
3345:         instead of question_context_external (which comes from orchestrator).
3346:         """
3347:         # Extract identity from contract
3348:         identity = contract["identity"]
3349:         base_slot = identity["base_slot"]
3350:         question_id = identity["question_id"]
3351:         dimension_id = identity["dimension_id"]
3352:         policy_area_id = identity["policy_area_id"]
3353:
3354:         # CALIBRATION ENFORCEMENT: Verify calibration status before execution
3355:         calibration = contract.get("calibration", {})
3356:         calibration_status = calibration.get("status", "placeholder")
3357:         if calibration_status == "placeholder":
3358:             abort_on_placeholder = (
3359:                 self.config.get("abort_on_placeholder_calibration", True)
3360:                 if hasattr(self.config, "get")
```

```
                else True
            )
            if abort_on_placeholder:
                note = calibration.get("note", "No calibration note provided")
                raise RuntimeError(
                    f"Contract {base_slot} has placeholder calibration (status={calibration_status}). "
                    f"Execution aborted per policy. Calibration note: {note}"
                )

        # Extract question context from contract (source of truth for v3)
        question_context = contract["question_context"]
        question_global = question_context_external.get(
            "question_global"
        )  # May come from orchestrator
        patterns = question_context.get("patterns", [])
        expected_elements = question_context.get("expected_elements", [])

        # Signal pack
        signal_pack = None
        if (
            self.signal_registry is not None
            and hasattr(self.signal_registry, "get")
            and policy_area_id
        ):
            signal_pack = self.signal_registry.get(policy_area_id)

        # SIGNAL REQUIREMENTS VALIDATION: Verify signal requirements from contract
        signal_requirements = contract.get("signal_requirements", {})
        if signal_requirements:
            self._validate_signal_requirements(
                signal_pack, signal_requirements, base_slot
            )

        # Extract method binding
        method_binding = contract["method_binding"]
        orchestration_mode = method_binding.get("orchestration_mode", "single_method")

        # Prepare common kwargs
        common_kwargs: dict[str, Any] = {
            "document": document,
            "base_slot": base_slot,
            "raw_text": getattr(document, "raw_text", None),
            "text": getattr(document, "raw_text", None),
            "question_id": question_id,
            "question_global": question_global,
            "policy_area_id": policy_area_id,
            "dimension_id": dimension_id,
            "cluster_id": identity.get("cluster_id"),
            "signal_pack": signal_pack,
            "question_patterns": patterns,
            "expected_elements": expected_elements,
            "question_context": question_context,
        }

        # Execute methods based on orchestration mode
        method_outputs: dict[str, Any] = {}
```

```
        signal_usage_list: list[dict[str, Any]] = []

        if orchestration_mode == "multi_method_pipeline":
            # Multi-method execution: process all methods in priority order
            methods = method_binding.get("methods", [])
            if not methods:
                raise ValueError(
                    f"orchestration_mode is 'multi_method_pipeline' but no methods array found in method_binding for {base_slot}"
                )

            # Sort by priority (lower priority number = execute first)
            sorted_methods = sorted(methods, key=lambda m: m.get("priority", 99))

            for method_spec in sorted_methods:
                class_name = method_spec["class_name"]
                method_name = method_spec["method_name"]
                provides = method_spec.get("provides", f"{class_name}.{method_name}")
                priority = method_spec.get("priority", 99)

                try:
                    result = self.method_executor.execute(
                        class_name=class_name,
                        method_name=method_name,
                        **common_kwargs,
                    )

                    # Store result using nested key structure (e.g., "text_mining.critical_links")
                    self._set_nested_value(method_outputs, provides, result)

                    # Track signal usage for this method
                    if signal_pack is not None:
                        signal_usage_list.append(
                            {
                                "method": f"{class_name}.{method_name}",
                                "policy_area": signal_pack.policy_area,
                                "version": signal_pack.version,
                                "priority": priority,
                            }
                        )

                except Exception as exc:
                    import logging

                    logging.error(
                        f"Method execution failed in multi-method pipeline: {class_name}.{method_name}",
                        exc_info=True,
                    )
                    # Store error in trace for debugging
                    # Store error in a flat structure under _errors[provides]
                    if "_errors" not in method_outputs or not isinstance(
                        method_outputs["_errors"], dict
                    ):
                        method_outputs["_errors"] = {}
                    method_outputs["_errors"][provides] = {
                        "error": str(exc),
                        "method": f"{class_name}.{method_name}",
```

```
                    }
                    # Re-raise if error_handling policy requires it
                    error_handling = contract.get("error_handling", {})
                    on_method_failure = error_handling.get(
                        "on_method_failure", "propagate_with_trace"
                    )
                    if on_method_failure == "raise":
                        raise
                    # Otherwise continue with other methods

        else:
            # Single-method execution (backward compatible, default)
            class_name = method_binding.get("class_name")
            method_name = method_binding.get("method_name")

            if not class_name or not method_name:
                # Try primary_method if direct class_name/method_name not found
                primary_method = method_binding.get("primary_method", {})
                class_name = primary_method.get("class_name") or class_name
                method_name = primary_method.get("method_name") or method_name

            if not class_name or not method_name:
                raise ValueError(
                    f"Invalid method_binding for {base_slot}: missing class_name or method_name"
                )

            result = self.method_executor.execute(
                class_name=class_name,
                method_name=method_name,
                **common_kwargs,
            )
            method_outputs["primary_analysis"] = result

            # Track signal usage
            if signal_pack is not None:
                signal_usage_list.append(
                    {
                        "method": f"{class_name}.{method_name}",
                        "policy_area": signal_pack.policy_area,
                        "version": signal_pack.version,
                    }
                )

        # Store signal usage in method_outputs for trace
        if signal_usage_list:
            method_outputs["_signal_usage"] = signal_usage_list

        # Evidence assembly
        evidence_assembly = contract["evidence_assembly"]
        assembly_rules = evidence_assembly["assembly_rules"]
        assembled = EvidenceAssembler.assemble(method_outputs, assembly_rules)
        evidence = assembled["evidence"]
        trace = assembled["trace"]

        # Validation with ENHANCED NA POLICY SUPPORT
        validation_rules_section = contract["validation_rules"]
```

```
        validation_rules = validation_rules_section.get("rules", [])
        na_policy = validation_rules_section.get("na_policy", "abort_on_critical")
        validation_rules_object = {"rules": validation_rules, "na_policy": na_policy}
        validation = EvidenceValidator.validate(evidence, validation_rules_object)

        # CONTRACT VALIDATION with ValidationOrchestrator
        contract_validation = None
        if self._use_validation_orchestrator:
            from farfan_pipeline.core.orchestrator.signal_contract_validator import (
                validate_result_with_orchestrator,
            )

            signal_node_for_validation = {
                "id": question_id,
                "failure_contract": error_handling.get("failure_contract", {}),
                "validations": validation_rules_object,
                "expected_elements": expected_elements
            }

            contract_validation = validate_result_with_orchestrator(
                result=evidence,
                signal_node=signal_node_for_validation,
                orchestrator=self.validation_orchestrator,
                auto_register=True
            )

            # Merge contract validation failures into standard validation
            if not contract_validation.passed:
                validation["contract_validation_failed"] = True
                validation["contract_error_code"] = contract_validation.error_code
                validation["contract_remediation"] = contract_validation.remediation
                validation["contract_failures"] = [
                    {
                        "type": f.failure_type,
                        "field": f.field_name,
                        "message": f.message,
                        "severity": f.severity
                    }
                    for f in contract_validation.failures_detailed[:10]
                ]

        # Handle validation failures based on NA policy
        validation_passed = validation.get("passed", True)
        if not validation_passed:
            if na_policy == "abort_on_critical":
                # Error handling will check failure contract below
                pass  # Let error_handling section handle abort
            elif na_policy == "score_zero":
                # Mark result as failed with score zero
                validation["score"] = 0.0
                validation["quality_level"] = "FAILED_VALIDATION"
                validation["na_policy_applied"] = "score_zero"
            elif na_policy == "propagate":
                # Continue with validation errors in result
                validation["na_policy_applied"] = "propagate"
                validation["validation_failed"] = True
```

```
3585:
3586:         # Error handling
3587:         error_handling = contract["error_handling"]
3588:         if error_handling:
3589:             evidence_with_validation = {**evidence, "validation": validation}
3590:             self._check_failure_contract(evidence_with_validation, error_handling)
3591:
3592:         # Build result
3593:         result_data = {
3594:             "base_slot": base_slot,
3595:             "question_id": question_id,
3596:             "question_global": question_global,
3597:             "policy_area_id": policy_area_id,
3598:             "dimension_id": dimension_id,
3599:             "cluster_id": identity.get("cluster_id"),
3600:             "evidence": evidence,
3601:             "validation": validation,
3602:             "trace": trace,
3603:             # CONTRACT VALIDATION METADATA
3604:             "contract_validation": {
3605:                 "enabled": contract_validation is not None,
3606:                 "passed": contract_validation.passed if contract_validation else None,
3607:                 "error_code": contract_validation.error_code if contract_validation else None,
3608:                 "failure_count": len(contract_validation.failures_detailed) if contract_validation else 0,
3609:                 "orchestrator_registered": self._use_validation_orchestrator
3610:             }
3611:         }
3612:
3613:         # Record evidence in global registry for provenance tracking
3614:         registry = get_global_registry()
3615:         registry.record_evidence(
3616:             evidence_type="executor_result_v3",
3617:             payload=result_data,
3618:             source_method=f"{self.__class__.__module__}.{self.__class__.__name__}.execute",
3619:             question_id=question_id,
3620:             document_id=getattr(document, "document_id", None),
3621:         )
3622:
3623:         # Validate output against output_contract schema if present
3624:         output_contract = contract.get("output_contract", {})
3625:         if output_contract and "schema" in output_contract:
3626:             self._validate_output_contract(
3627:                 result_data, output_contract["schema"], base_slot
3628:             )
3629:
3630:         # Generate human_readable_output if template exists
3631:         human_readable_config = output_contract.get("human_readable_output", {})
3632:         if human_readable_config:
3633:             result_data["human_readable_output"] = self._generate_human_readable_output(
3634:                 evidence, validation, human_readable_config, contract
3635:             )
3636:
3637:         return result_data
3638:
3639:     def _validate_output_contract(
3640:         self, result: dict[str, Any], schema: dict[str, Any], base_slot: str
```

```
    ) -> None:
        """Validate result against output_contract schema with detailed error messages.

        Args:
            result: Result data to validate
            schema: JSON Schema from contract
            base_slot: Base slot identifier for error messages

        Raises:
            ValueError: If validation fails with detailed path information
        """
        from jsonschema import ValidationError, validate

        try:
            validate(instance=result, schema=schema)
        except ValidationError as e:
            # Enhanced error message with JSON path
            path = (
                ".".join(str(p) for p in e.absolute_path) if e.absolute_path else "root"
            )
            raise ValueError(
                f"Output contract validation failed for {base_slot} at '{path}': {e.message}. "
                f"Schema constraint: {e.schema}"
            ) from e

    def _generate_human_readable_output(
        self,
        evidence: dict[str, Any],
        validation: dict[str, Any],
        config: dict[str, Any],
        contract: dict[str, Any],
    ) -> str:
        """Generate production-grade human-readable output from template.

        Implements full template engine with:
        - Variable substitution with dot-notation: {evidence.elements_found_count}
        - Derived metrics: Automatic calculation of means, counts, percentages
        - List formatting: Convert arrays to markdown/html/plain_text lists
        - Methodological depth rendering: Full epistemological documentation
        - Multi-format support: markdown, html, plain_text with proper formatting

        Args:
            evidence: Evidence dict from executor
            validation: Validation dict
            config: human_readable_output config from contract
            contract: Full contract for methodological_depth access

        Returns:
            Formatted string in specified format
        """
        template_config = config.get("template", {})
        format_type = config.get("format", "markdown")
        methodological_depth_config = config.get("methodological_depth", {})

        # Build context for variable substitution
        context = self._build_template_context(evidence, validation, contract)
```

```
3697:
3698:         # Render each template section
3699:         sections = []
3700:
3701:         # Title
3702:         if "title" in template_config:
3703:             sections.append(
3704:                 self._render_template_string(
3705:                     template_config["title"], context, format_type
3706:                 )
3707:             )
3708:
3709:         # Summary
3710:         if "summary" in template_config:
3711:             sections.append(
3712:                 self._render_template_string(
3713:                     template_config["summary"], context, format_type
3714:                 )
3715:             )
3716:
3717:         # Score section
3718:         if "score_section" in template_config:
3719:             sections.append(
3720:                 self._render_template_string(
3721:                     template_config["score_section"], context, format_type
3722:                 )
3723:             )
3724:
3725:         # Elements section
3726:         if "elements_section" in template_config:
3727:             sections.append(
3728:                 self._render_template_string(
3729:                     template_config["elements_section"], context, format_type
3730:                 )
3731:             )
3732:
3733:         # Details (list of items)
3734:         if "details" in template_config and isinstance(
3735:             template_config["details"], list
3736:         ):
3737:             detail_items = [
3738:                 self._render_template_string(item, context, format_type)
3739:                 for item in template_config["details"]
3740:             ]
3741:             sections.append(self._format_list(detail_items, format_type))
3742:
3743:         # Interpretation
3744:         if "interpretation" in template_config:
3745:             # Add methodological interpretation if available
3746:             context["methodological_interpretation"] = (
3747:                 self._render_methodological_depth(
3748:                     methodological_depth_config, evidence, validation, format_type
3749:                 )
3750:             )
3751:             sections.append(
3752:                 self._render_template_string(
```

```
                    template_config["interpretation"], context, format_type
                )
            )

        # Recommendations
        if "recommendations" in template_config:
            sections.append(
                self._render_template_string(
                    template_config["recommendations"], context, format_type
                )
            )

        # Join sections with appropriate separator for format
        separator = (
            "\n\n"
            if format_type == "markdown"
            else "\n\n" if format_type == "plain_text" else "<br><br>"
        )
        return separator.join(filter(None, sections))

    def _build_template_context(
        self,
        evidence: dict[str, Any],
        validation: dict[str, Any],
        contract: dict[str, Any],
    ) -> dict[str, Any]:
        """Build comprehensive context for template variable substitution.

        Args:
            evidence: Evidence dict
            validation: Validation dict
            contract: Full contract

        Returns:
            Context dict with all variables and derived metrics
        """
        # Base context
        context = {
            "evidence": evidence.copy(),
            "validation": validation.copy(),
        }

        # Add derived metrics from evidence
        if "elements" in evidence and isinstance(evidence["elements"], list):
            context["evidence"]["elements_found_count"] = len(evidence["elements"])
            context["evidence"]["elements_found_list"] = self._format_evidence_list(
                evidence["elements"]
            )

        if "confidences" in evidence and isinstance(evidence["confidences"], list):
            confidences = evidence["confidences"]
            if confidences:
                context["evidence"]["confidence_scores"] = {
                    "mean": sum(confidences) / len(confidences),
                    "min": min(confidences),
                    "max": max(confidences),
```

```
3809:                 }
3810:
3811:         if "patterns" in evidence and isinstance(evidence["patterns"], dict):
3812:             context["evidence"]["pattern_matches_count"] = len(evidence["patterns"])
3813:
3814:         # Add defaults for missing keys to prevent KeyError
3815:         context["evidence"].setdefault("missing_required_elements", "None")
3816:         context["evidence"].setdefault("official_sources_count", 0)
3817:         context["evidence"].setdefault("quantitative_indicators_count", 0)
3818:         context["evidence"].setdefault("temporal_series_count", 0)
3819:         context["evidence"].setdefault("territorial_coverage", "Not specified")
3820:         context["evidence"].setdefault(
3821:             "recommendations", "No specific recommendations available"
3822:         )
3823:
3824:         # Add score and quality from validation or defaults
3825:         context["score"] = validation.get("score", 0.0)
3826:         context["quality_level"] = self._determine_quality_level(
3827:             validation.get("score", 0.0)
3828:         )
3829:
3830:         return context
3831:
3832:     def _determine_quality_level(self, score: float) -> str:
3833:         """Determine quality level from score.
3834:
3835:         Args:
3836:             score: Numeric score (typically 0.0-3.0)
3837:
3838:         Returns:
3839:             Quality level string
3840:         """
3841:         if score >= 2.5:
3842:             return "EXCELLENT"
3843:         elif score >= 2.0:
3844:             return "GOOD"
3845:         elif score >= 1.0:
3846:             return "ACCEPTABLE"
3847:         elif score > 0:
3848:             return "INSUFFICIENT"
3849:         else:
3850:             return "FAILED"
3851:
3852:     def _render_template_string(
3853:         self, template: str, context: dict[str, Any], format_type: str
3854:     ) -> str:
3855:         """Render a template string with variable substitution.
3856:
3857:         Supports dot-notation: {evidence.elements_found_count}
3858:         Supports arithmetic: {score}/3.0 (rendered as-is, user interprets)
3859:
3860:         Args:
3861:             template: Template string with {variable} placeholders
3862:             context: Context dict
3863:             format_type: Output format (markdown, html, plain_text)
3864:
```

```
        Returns:
            Rendered string with variables substituted
        """
        import re

        def replace_var(match):
            var_path = match.group(1)
            try:
                # Handle dot-notation traversal
                keys = var_path.split(".")
                value = context
                for key in keys:
                    if isinstance(value, dict):
                        value = value[key]
                    else:
                        # Try to get attribute (for objects)
                        value = getattr(value, key, None)
                        if value is None:
                            return f"{{MISSING:{var_path}}}"

                # Format value appropriately
                if isinstance(value, float):
                    return f"{value:.2f}"
                elif isinstance(value, list | dict):
                    return str(value)  # Simple representation
                else:
                    return str(value)
            except (KeyError, AttributeError, TypeError):
                return f"{{MISSING:{var_path}}}"

        # Replace all {variable} patterns
        rendered = re.sub(r"\{([^}]+)\}", replace_var, template)
        return rendered

    def _format_evidence_list(self, elements: list) -> str:
        """Format evidence elements as markdown list.

        Args:
            elements: List of evidence elements

        Returns:
            Markdown-formatted list string
        """
        if not elements:
            return "- No elements found"

        formatted = []
        for elem in elements:
            if isinstance(elem, dict):
                # Try to extract meaningful representation
                elem_str = elem.get("description") or elem.get("type") or str(elem)
            else:
                elem_str = str(elem)
            formatted.append(f"- {elem_str}")

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

```
3921:
3922:     def _format_list(self, items: list[str], format_type: str) -> str:
3923:         """Format a list of items according to output format.
3924:
3925:         Args:
3926:             items: List of string items
3927:             format_type: Output format
3928:
3929:         Returns:
3930:             Formatted list string
3931:         """
3932:         if format_type == "html":
3933:             items_html = "".join(f"<li>{item}</li>" for item in items)
3934:             return f"<ul>{items_html}</ul>"
3935:         else:  # markdown or plain_text
3936:             return "\n".join(f"- {item}" for item in items)
3937:
3938:     def _render_methodological_depth(
3939:         self,
3940:         config: dict[str, Any],
3941:         evidence: dict[str, Any],
3942:         validation: dict[str, Any],
3943:         format_type: str,
3944:     ) -> str:
3945:         """Render methodological depth section with epistemological foundations.
3946:
3947:         Transforms v3 contract's methodological_depth into comprehensive documentation.
3948:
3949:         Args:
3950:             config: methodological_depth config from contract
3951:             evidence: Evidence dict for contextualization
3952:             validation: Validation dict
3953:             format_type: Output format
3954:
3955:         Returns:
3956:             Formatted methodological depth documentation
3957:         """
3958:         if not config or "methods" not in config:
3959:             return "Methodological documentation not available for this executor."
3960:
3961:         sections = []
3962:
3963:         # Header
3964:         if format_type == "markdown":
3965:             sections.append("#### Methodological Foundations\n")
3966:         elif format_type == "html":
3967:             sections.append("<h4>Methodological Foundations</h4>")
3968:         else:
3969:             sections.append("METHODOLOGICAL FOUNDATIONS\n")
3970:
3971:         methods = config.get("methods", [])
3972:
3973:         for method_info in methods:
3974:             method_name = method_info.get("method_name", "Unknown")
3975:             class_name = method_info.get("class_name", "Unknown")
3976:             priority = method_info.get("priority", 0)
```

```
3977:             role = method_info.get("role", "analysis")
3978:
3979:             # Method header
3980:             if format_type == "markdown":
3981:                 sections.append(
3982:                     f"##### {class_name}.{method_name} (Priority {priority}, Role: {role})\n"
3983:                 )
3984:             else:
3985:                 sections.append(
3986:                     f"\n{class_name}.{method_name} (Priority {priority}, Role: {role})\n"
3987:                 )
3988:
3989:             # Epistemological foundation
3990:             epist = method_info.get("epistemological_foundation", {})
3991:             if epist:
3992:                 sections.append(
3993:                     self._render_epistemological_foundation(epist, format_type)
3994:                 )
3995:
3996:             # Technical approach
3997:             technical = method_info.get("technical_approach", {})
3998:             if technical:
3999:                 sections.append(self._render_technical_approach(technical, format_type))
4000:
4001:             # Output interpretation
4002:             output_interp = method_info.get("output_interpretation", {})
4003:             if output_interp:
4004:                 sections.append(
4005:                     self._render_output_interpretation(output_interp, format_type)
4006:                 )
4007:
4008:         # Method combination logic
4009:         combination = config.get("method_combination_logic", {})
4010:         if combination:
4011:             sections.append(self._render_method_combination(combination, format_type))
4012:
4013:         return "\n\n".join(filter(None, sections))
4014:
4015:     def _render_epistemological_foundation(
4016:         self, foundation: dict[str, Any], format_type: str
4017:     ) -> str:
4018:         """Render epistemological foundation section.
4019:
4020:         Args:
4021:             foundation: Epistemological foundation dict
4022:             format_type: Output format
4023:
4024:         Returns:
4025:             Formatted epistemological foundation text
4026:         """
4027:         parts = []
4028:
4029:         paradigm = foundation.get("paradigm")
4030:         if paradigm:
4031:             parts.append(f"**Paradigm**: {paradigm}")
4032:
```

```
        ontology = foundation.get("ontological_basis")
        if ontology:
            parts.append(f"**Ontological Basis**: {ontology}")

        stance = foundation.get("epistemological_stance")
        if stance:
            parts.append(f"**Epistemological Stance**: {stance}")

        framework = foundation.get("theoretical_framework", [])
        if framework:
            parts.append("**Theoretical Framework**:")
            for item in framework:
                parts.append(f"  - {item}")

        justification = foundation.get("justification")
        if justification:
            parts.append(f"**Justification**: {justification}")

        return "\n".join(parts) if format_type != "html" else "<br>".join(parts)

    def _render_technical_approach(
        self, technical: dict[str, Any], format_type: str
    ) -> str:
        """Render technical approach section.

        Args:
            technical: Technical approach dict
            format_type: Output format

        Returns:
            Formatted technical approach text
        """
        parts = []

        method_type = technical.get("method_type")
        if method_type:
            parts.append(f"**Method Type**: {method_type}")

        algorithm = technical.get("algorithm")
        if algorithm:
            parts.append(f"**Algorithm**: {algorithm}")

        steps = technical.get("steps", [])
        if steps:
            parts.append("**Processing Steps**:")
            for step in steps:
                step_num = step.get("step", "?")
                step_name = step.get("name", "Unnamed")
                step_desc = step.get("description", "")
                parts.append(f"  {step_num}. **{step_name}**: {step_desc}")

        assumptions = technical.get("assumptions", [])
        if assumptions:
            parts.append("**Assumptions**:")
            for assumption in assumptions:
                parts.append(f"  - {assumption}")
```

```
4089:
4090:         limitations = technical.get("limitations", [])
4091:         if limitations:
4092:             parts.append("**Limitations**:")
4093:             for limitation in limitations:
4094:                 parts.append(f"  - {limitation}")
4095:
4096:         return "\n".join(parts) if format_type != "html" else "<br>".join(parts)
4097:
4098:     def _render_output_interpretation(
4099:         self, interpretation: dict[str, Any], format_type: str
4100:     ) -> str:
4101:         """Render output interpretation section.
4102:
4103:         Args:
4104:             interpretation: Output interpretation dict
4105:             format_type: Output format
4106:
4107:         Returns:
4108:             Formatted output interpretation text
4109:         """
4110:         parts = []
4111:
4112:         guide = interpretation.get("interpretation_guide", {})
4113:         if guide:
4114:             parts.append("**Interpretation Guide**:")
4115:             for threshold_name, threshold_desc in guide.items():
4116:                 parts.append(f"  - **{threshold_name}**: {threshold_desc}")
4117:
4118:         insights = interpretation.get("actionable_insights", [])
4119:         if insights:
4120:             parts.append("**Actionable Insights**:")
4121:             for insight in insights:
4122:                 parts.append(f"  - {insight}")
4123:
4124:         return "\n".join(parts) if format_type != "html" else "<br>".join(parts)
4125:
4126:     def _render_method_combination(
4127:         self, combination: dict[str, Any], format_type: str
4128:     ) -> str:
4129:         """Render method combination logic section.
4130:
4131:         Args:
4132:             combination: Method combination dict
4133:             format_type: Output format
4134:
4135:         Returns:
4136:             Formatted method combination text
4137:         """
4138:         parts = []
4139:
4140:         if format_type == "markdown":
4141:             parts.append("#### Method Combination Strategy\n")
4142:         else:
4143:             parts.append("METHOD COMBINATION STRATEGY\n")
4144:
```

```
        strategy = combination.get("combination_strategy")
        if strategy:
            parts.append(f"**Strategy**: {strategy}")

        rationale = combination.get("rationale")
        if rationale:
            parts.append(f"**Rationale**: {rationale}")

        fusion = combination.get("evidence_fusion")
        if fusion:
            parts.append(f"**Evidence Fusion**: {fusion}")

        return "\n".join(parts) if format_type != "html" else "<br>".join(parts)


================================================================================
FILE: src/farfan_pipeline/core/orchestrator/batch_executor.py
================================================================================

"""Batch processing infrastructure for executor scalability.

This module provides batched entity processing with:
- Configurable batch sizes per executor type based on object complexity
- Streaming result aggregation to avoid memory accumulation
- Async executor flow integration for parallel batch processing
- Batch-level error handling with partial success recovery

Example:
    >>> config = BatchExecutorConfig(default_batch_size=10, max_batch_size=100)
    >>> executor = BatchExecutor(config, method_executor, signal_registry)
    >>> async for batch_result in executor.execute_batches(entities, question_context):
    ...     process_batch_result(batch_result)
"""

from __future__ import annotations

import asyncio
import logging
import time
from collections.abc import AsyncIterator, Callable, Iterable
from dataclasses import dataclass, field
from enum import Enum
from typing import TYPE_CHECKING, Any, TypedDict

if TYPE_CHECKING:
    from farfan_pipeline.core.orchestrator.core import MethodExecutor
    from farfan_pipeline.core.types import PreprocessedDocument

logger = logging.getLogger(__name__)


class BatchStatus(Enum):
    """Status of batch execution."""

    PENDING = "pending"
```

```
4201:     RUNNING = "running"
4202:     COMPLETED = "completed"
4203:     PARTIAL_SUCCESS = "partial_success"
4204:     FAILED = "failed"
4205:     CANCELLED = "cancelled"
4206:
4207:
4208: class ExecutorComplexity(Enum):
4209:     """Complexity classification for executor types."""
4210:
4211:     SIMPLE = "simple"
4212:     MODERATE = "moderate"
4213:     COMPLEX = "complex"
4214:     VERY_COMPLEX = "very_complex"
4215:
4216:
4217: @dataclass
4218: class BatchExecutorConfig:
4219:     """Configuration for batch processing.
4220:
4221:     Attributes:
4222:         default_batch_size: Default batch size for unclassified executors
4223:         max_batch_size: Maximum batch size allowed
4224:         min_batch_size: Minimum batch size allowed
4225:         enable_streaming: Enable streaming result aggregation
4226:         error_threshold: Fraction of failures before marking batch as failed (0.0-1.0)
4227:         max_retries: Maximum number of retries for failed batches
4228:         backoff_base_seconds: Base delay for exponential backoff
4229:         enable_instrumentation: Enable detailed logging and metrics
4230:     """
4231:
4232:     default_batch_size: int = 10
4233:     max_batch_size: int = 100
4234:     min_batch_size: int = 1
4235:     enable_streaming: bool = True
4236:     error_threshold: float = 0.5
4237:     max_retries: int = 2
4238:     backoff_base_seconds: float = 1.0
4239:     enable_instrumentation: bool = True
4240:
4241:     def __post_init__(self) -> None:
4242:         """Validate configuration parameters."""
4243:         if not (
4244:             1 <= self.min_batch_size <= self.default_batch_size <= self.max_batch_size
4245:         ):
4246:             raise ValueError(
4247:                 f"Invalid batch size configuration: min={self.min_batch_size}, "
4248:                 f"default={self.default_batch_size}, max={self.max_batch_size}"
4249:             )
4250:         if not 0.0 <= self.error_threshold <= 1.0:
4251:             raise ValueError(
4252:                 f"error_threshold must be in [0.0, 1.0], got {self.error_threshold}"
4253:             )
4254:
4255:
4256: EXECUTOR_COMPLEXITY_MAP: dict[str, ExecutorComplexity] = {
```

```
4257:     "D1-Q1": ExecutorComplexity.MODERATE,
4258:     "D1-Q2": ExecutorComplexity.MODERATE,
4259:     "D1-Q3": ExecutorComplexity.COMPLEX,
4260:     "D1-Q4": ExecutorComplexity.MODERATE,
4261:     "D1-Q5": ExecutorComplexity.MODERATE,
4262:     "D2-Q1": ExecutorComplexity.MODERATE,
4263:     "D2-Q2": ExecutorComplexity.VERY_COMPLEX,
4264:     "D2-Q3": ExecutorComplexity.COMPLEX,
4265:     "D2-Q4": ExecutorComplexity.COMPLEX,
4266:     "D2-Q5": ExecutorComplexity.MODERATE,
4267:     "D3-Q1": ExecutorComplexity.MODERATE,
4268:     "D3-Q2": ExecutorComplexity.VERY_COMPLEX,
4269:     "D3-Q3": ExecutorComplexity.VERY_COMPLEX,
4270:     "D3-Q4": ExecutorComplexity.VERY_COMPLEX,
4271:     "D3-Q5": ExecutorComplexity.VERY_COMPLEX,
4272:     "D4-Q1": ExecutorComplexity.VERY_COMPLEX,
4273:     "D4-Q2": ExecutorComplexity.COMPLEX,
4274:     "D4-Q3": ExecutorComplexity.COMPLEX,
4275:     "D4-Q4": ExecutorComplexity.MODERATE,
4276:     "D4-Q5": ExecutorComplexity.MODERATE,
4277:     "D5-Q1": ExecutorComplexity.MODERATE,
4278:     "D5-Q2": ExecutorComplexity.VERY_COMPLEX,
4279:     "D5-Q3": ExecutorComplexity.MODERATE,
4280:     "D5-Q4": ExecutorComplexity.COMPLEX,
4281:     "D5-Q5": ExecutorComplexity.COMPLEX,
4282:     "D6-Q1": ExecutorComplexity.COMPLEX,
4283:     "D6-Q2": ExecutorComplexity.MODERATE,
4284:     "D6-Q3": ExecutorComplexity.COMPLEX,
4285:     "D6-Q4": ExecutorComplexity.MODERATE,
4286:     "D6-Q5": ExecutorComplexity.MODERATE,
4287: }
4288:
4289: COMPLEXITY_BATCH_SIZE_MAP: dict[ExecutorComplexity, int] = {
4290:     ExecutorComplexity.SIMPLE: 50,
4291:     ExecutorComplexity.MODERATE: 20,
4292:     ExecutorComplexity.COMPLEX: 10,
4293:     ExecutorComplexity.VERY_COMPLEX: 5,
4294: }
4295:
4296:
4297: class BatchMetrics(TypedDict):
4298:     """Metrics for a single batch execution."""
4299:
4300:     batch_id: str
4301:     batch_index: int
4302:     batch_size: int
4303:     status: BatchStatus
4304:     start_time: float
4305:     end_time: float | None
4306:     execution_time_ms: float
4307:     successful_items: int
4308:     failed_items: int
4309:     retries_used: int
4310:     error_messages: list[str]
4311:
4312:
```

```
@dataclass
class BatchResult:
    """Result of a batch execution.

    Attributes:
        batch_id: Unique batch identifier
        batch_index: Index of the batch in the sequence
        items: List of items in this batch
        results: List of results corresponding to items (may contain None for failures)
        status: Batch execution status
        metrics: Execution metrics
        errors: List of errors encountered (empty if successful)
    """

    batch_id: str
    batch_index: int
    items: list[Any]
    results: list[Any]
    status: BatchStatus
    metrics: BatchMetrics
    errors: list[dict[str, Any]] = field(default_factory=list)

    def is_successful(self) -> bool:
        """Check if batch execution was fully successful."""
        return self.status == BatchStatus.COMPLETED and not self.errors

    def has_partial_success(self) -> bool:
        """Check if batch had partial success."""
        return self.status == BatchStatus.PARTIAL_SUCCESS and any(
            r is not None for r in self.results
        )

    def get_successful_results(self) -> list[tuple[Any, Any]]:
        """Get list of (item, result) pairs for successful executions."""
        return [
            (item, result)
            for item, result in zip(self.items, self.results, strict=False)
            if result is not None
        ]

    def get_failed_items(self) -> list[Any]:
        """Get list of items that failed execution."""
        return [
            item
            for item, result in zip(self.items, self.results, strict=False)
            if result is None
        ]


@dataclass
class AggregatedBatchResults:
    """Aggregated results from multiple batch executions.

    Attributes:
        total_batches: Total number of batches processed
        total_items: Total number of items processed
```

```
4369:         successful_items: Number of successfully processed items
4370:         failed_items: Number of failed items
4371:         results: List of all successful results
4372:         errors: List of all errors encountered
4373:         execution_time_ms: Total execution time in milliseconds
4374:         batch_metrics: List of metrics for each batch
4375:     """
4376:
4377:     total_batches: int
4378:     total_items: int
4379:     successful_items: int
4380:     failed_items: int
4381:     results: list[Any]
4382:     errors: list[dict[str, Any]]
4383:     execution_time_ms: float
4384:     batch_metrics: list[BatchMetrics]
4385:
4386:     def success_rate(self) -> float:
4387:         """Calculate success rate as fraction of successful items."""
4388:         if self.total_items == 0:
4389:             return 0.0
4390:         return self.successful_items / self.total_items
4391:
4392:
4393: class BatchExecutor:
4394:     """Batch processing executor with streaming aggregation and error recovery.
4395:
4396:     This executor provides scalable batch processing for entity collections with:
4397:     - Adaptive batch sizing based on executor complexity
4398:     - Streaming result aggregation to avoid memory accumulation
4399:     - Parallel batch processing via async executor flow
4400:     - Batch-level error handling with partial success recovery
4401:     """
4402:
4403:     def __init__(
4404:         self,
4405:         config: BatchExecutorConfig | None = None,
4406:         method_executor: MethodExecutor | None = None,
4407:         signal_registry: Any | None = None,
4408:         questionnaire_provider: Any | None = None,
4409:         calibration_orchestrator: Any | None = None,
4410:     ) -> None:
4411:         """Initialize batch executor.
4412:
4413:         Args:
4414:             config: Batch execution configuration
4415:             method_executor: MethodExecutor instance for method routing
4416:             signal_registry: Signal registry for executor instances
4417:             questionnaire_provider: Questionnaire provider
4418:             calibration_orchestrator: Calibration orchestrator
4419:         """
4420:         self.config = config or BatchExecutorConfig()
4421:         self.method_executor = method_executor
4422:         self.signal_registry = signal_registry
4423:         self.questionnaire_provider = questionnaire_provider
4424:         self.calibration_orchestrator = calibration_orchestrator
```

```
4425:         self._batch_counter = 0
4426:
4427:         if self.config.enable_instrumentation:
4428:             logger.info(
4429:                 f"BatchExecutor initialized: default_batch_size={self.config.default_batch_size}, "
4430:                 f"max_batch_size={self.config.max_batch_size}, "
4431:                 f"error_threshold={self.config.error_threshold}"
4432:             )
4433:
4434:     def get_batch_size_for_executor(self, base_slot: str) -> int:
4435:         """Determine batch size for a given executor based on complexity.
4436:
4437:         Args:
4438:             base_slot: Executor base slot (e.g., "D1-Q1")
4439:
4440:         Returns:
4441:             Recommended batch size for this executor
4442:         """
4443:         complexity = EXECUTOR_COMPLEXITY_MAP.get(base_slot, ExecutorComplexity.MODERATE)
4444:         recommended_size = COMPLEXITY_BATCH_SIZE_MAP.get(
4445:             complexity, self.config.default_batch_size
4446:         )
4447:
4448:         batch_size = max(
4449:             self.config.min_batch_size,
4450:             min(recommended_size, self.config.max_batch_size),
4451:         )
4452:
4453:         if self.config.enable_instrumentation:
4454:             logger.debug(
4455:                 f"Batch size for {base_slot}: {batch_size} "
4456:                 f"(complexity={complexity.value}, recommended={recommended_size})"
4457:             )
4458:
4459:         return batch_size
4460:
4461:     def _create_batches(
4462:         self, items: list[Any], batch_size: int
4463:     ) -> list[tuple[int, list[Any]]]:
4464:         """Split items into batches of specified size.
4465:
4466:         Args:
4467:             items: List of items to batch
4468:             batch_size: Size of each batch
4469:
4470:         Returns:
4471:             List of (batch_index, batch_items) tuples
4472:         """
4473:         batches = []
4474:         for i in range(0, len(items), batch_size):
4475:             batch_index = i // batch_size
4476:             batch_items = items[i : i + batch_size]
4477:             batches.append((batch_index, batch_items))
4478:
4479:         if self.config.enable_instrumentation:
4480:             logger.info(
```

```
                f"Created {len(batches)} batches from {len(items)} items "
                f"(batch_size={batch_size})"
            )

        return batches

    async def _execute_single_batch(
        self,
        batch_id: str,
        batch_index: int,
        batch_items: list[Any],
        executor_instance: Any,
        document: PreprocessedDocument,
        question_context: dict[str, Any],
    ) -> BatchResult:
        """Execute a single batch of items.

        Args:
            batch_id: Unique batch identifier
            batch_index: Index of this batch
            batch_items: Items to process in this batch
            executor_instance: Executor instance to use
            document: Document being processed
            question_context: Question context for execution

        Returns:
            BatchResult with execution details
        """
        start_time = time.perf_counter()
        results: list[Any] = []
        errors: list[dict[str, Any]] = []
        successful_items = 0
        failed_items = 0

        if self.config.enable_instrumentation:
            logger.debug(
                f"[{batch_id}] Executing batch {batch_index} with {len(batch_items)} items"
            )

        for item_index, item in enumerate(batch_items):
            try:
                result = await asyncio.to_thread(
                    executor_instance.execute,
                    document,
                    self.method_executor,
                    question_context=question_context,
                )
                results.append(result)
                successful_items += 1
            except Exception as exc:
                results.append(None)
                failed_items += 1
                error_detail = {
                    "batch_id": batch_id,
                    "batch_index": batch_index,
                    "item_index": item_index,
```

```
                    "error_type": type(exc).__name__,
                    "error_message": str(exc),
                }
                errors.append(error_detail)

                if self.config.enable_instrumentation:
                    logger.warning(
                        f"[{batch_id}] Item {item_index} failed: {exc}", exc_info=False
                    )

        end_time = time.perf_counter()
        execution_time_ms = (end_time - start_time) * 1000.0

        error_rate = failed_items / len(batch_items) if batch_items else 0.0
        if error_rate >= self.config.error_threshold:
            status = BatchStatus.FAILED
        elif failed_items > 0:
            status = BatchStatus.PARTIAL_SUCCESS
        else:
            status = BatchStatus.COMPLETED

        metrics: BatchMetrics = {
            "batch_id": batch_id,
            "batch_index": batch_index,
            "batch_size": len(batch_items),
            "status": status,
            "start_time": start_time,
            "end_time": end_time,
            "execution_time_ms": execution_time_ms,
            "successful_items": successful_items,
            "failed_items": failed_items,
            "retries_used": 0,
            "error_messages": [e["error_message"] for e in errors],
        }

        if self.config.enable_instrumentation:
            logger.info(
                f"[{batch_id}] Batch {batch_index} completed: "
                f"{successful_items}/{len(batch_items)} successful "
                f"({execution_time_ms:.2f}ms, status={status.value})"
            )

        return BatchResult(
            batch_id=batch_id,
            batch_index=batch_index,
            items=batch_items,
            results=results,
            status=status,
            metrics=metrics,
            errors=errors,
        )

    async def _execute_batch_with_retry(
        self,
        batch_id: str,
        batch_index: int,
```

```
        batch_items: list[Any],
        executor_instance: Any,
        document: PreprocessedDocument,
        question_context: dict[str, Any],
    ) -> BatchResult:
        """Execute a batch with retry logic for failed batches.

        Args:
            batch_id: Unique batch identifier
            batch_index: Index of this batch
            batch_items: Items to process in this batch
            executor_instance: Executor instance to use
            document: Document being processed
            question_context: Question context for execution

        Returns:
            BatchResult with execution details
        """
        retry_count = 0
        last_result: BatchResult | None = None

        while retry_count <= self.config.max_retries:
            result = await self._execute_single_batch(
                batch_id,
                batch_index,
                batch_items,
                executor_instance,
                document,
                question_context,
            )

            if result.status in (BatchStatus.COMPLETED, BatchStatus.PARTIAL_SUCCESS):
                result.metrics["retries_used"] = retry_count
                return result

            last_result = result
            retry_count += 1

            if retry_count <= self.config.max_retries:
                backoff_delay = self.config.backoff_base_seconds * (
                    2 ** (retry_count - 1)
                )
                if self.config.enable_instrumentation:
                    logger.warning(
                        f"[{batch_id}] Batch {batch_index} failed (attempt {retry_count}), "
                        f"retrying after {backoff_delay:.2f}s"
                    )
                await asyncio.sleep(backoff_delay)

        if last_result:
            last_result.metrics["retries_used"] = retry_count - 1
            if self.config.enable_instrumentation:
                logger.error(
                    f"[{batch_id}] Batch {batch_index} failed after {retry_count - 1} retries"
                )
            return last_result
```

```
4649:
4650:         raise RuntimeError(f"Batch {batch_id} execution failed without result")
4651:
4652:     async def execute_batches(
4653:         self,
4654:         items: list[Any],
4655:         executor_class: type,
4656:         document: PreprocessedDocument,
4657:         question_context: dict[str, Any],
4658:         base_slot: str | None = None,
4659:         batch_size: int | None = None,
4660:     ) -> AsyncIterator[BatchResult]:
4661:         """Execute items in batches with streaming result generation.
4662:
4663:         This method processes items in batches and yields results as they complete,
4664:         enabling streaming aggregation without accumulating all results in memory.
4665:
4666:         Args:
4667:             items: List of items to process
4668:             executor_class: Executor class to instantiate
4669:             document: Document being processed
4670:             question_context: Question context for execution
4671:             base_slot: Base slot for complexity-based batch sizing (optional)
4672:             batch_size: Override batch size (optional, uses complexity-based sizing if None)
4673:
4674:         Yields:
4675:             BatchResult for each completed batch
4676:
4677:         Example:
4678:             >>> batches = executor.execute_batches(entities, D1Q1_Executor, doc, ctx)
4679:             >>> async for batch_result in batches:
4680:             ...     for item, result in batch_result.get_successful_results():
4681:             ...         aggregate(result)
4682:         """
4683:         if not items:
4684:             logger.warning("execute_batches called with empty items list")
4685:             return
4686:
4687:         if batch_size is None:
4688:             if base_slot:
4689:                 batch_size = self.get_batch_size_for_executor(base_slot)
4690:             else:
4691:                 batch_size = self.config.default_batch_size
4692:
4693:         batches = self._create_batches(items, batch_size)
4694:
4695:         for batch_index, batch_items in batches:
4696:             self._batch_counter += 1
4697:             batch_id = f"batch_{self._batch_counter:06d}"
4698:
4699:             executor_instance = executor_class(
4700:                 method_executor=self.method_executor,
4701:                 signal_registry=self.signal_registry,
4702:                 config=self.config,
4703:                 questionnaire_provider=self.questionnaire_provider,
4704:                 calibration_orchestrator=self.calibration_orchestrator,
```

```
            )

            batch_result = await self._execute_batch_with_retry(
                batch_id,
                batch_index,
                batch_items,
                executor_instance,
                document,
                question_context,
            )

            yield batch_result

    async def execute_batches_parallel(
        self,
        items: list[Any],
        executor_class: type,
        document: PreprocessedDocument,
        question_context: dict[str, Any],
        base_slot: str | None = None,
        batch_size: int | None = None,
        max_concurrent_batches: int = 4,
    ) -> list[BatchResult]:
        """Execute batches in parallel with concurrency control.

        Args:
            items: List of items to process
            executor_class: Executor class to instantiate
            document: Document being processed
            question_context: Question context for execution
            base_slot: Base slot for complexity-based batch sizing (optional)
            batch_size: Override batch size (optional)
            max_concurrent_batches: Maximum number of concurrent batch executions

        Returns:
            List of BatchResults for all batches
        """
        if not items:
            return []

        if batch_size is None:
            if base_slot:
                batch_size = self.get_batch_size_for_executor(base_slot)
            else:
                batch_size = self.config.default_batch_size

        batches = self._create_batches(items, batch_size)
        semaphore = asyncio.Semaphore(max_concurrent_batches)
        results: list[BatchResult] = []

        async def process_batch_with_semaphore(
            batch_index: int, batch_items: list[Any]
        ) -> BatchResult:
            async with semaphore:
                self._batch_counter += 1
                batch_id = f"batch_{self._batch_counter:06d}"
```

```
4761:
4762:                 executor_instance = executor_class(
4763:                     method_executor=self.method_executor,
4764:                     signal_registry=self.signal_registry,
4765:                     config=self.config,
4766:                     questionnaire_provider=self.questionnaire_provider,
4767:                     calibration_orchestrator=self.calibration_orchestrator,
4768:                 )
4769:
4770:                 return await self._execute_batch_with_retry(
4771:                     batch_id,
4772:                     batch_index,
4773:                     batch_items,
4774:                     executor_instance,
4775:                     document,
4776:                     question_context,
4777:                 )
4778:
4779:         tasks = [
4780:             asyncio.create_task(process_batch_with_semaphore(batch_index, batch_items))
4781:             for batch_index, batch_items in batches
4782:         ]
4783:
4784:         if self.config.enable_instrumentation:
4785:             logger.info(
4786:                 f"Executing {len(tasks)} batches in parallel "
4787:                 f"(max_concurrent={max_concurrent_batches})"
4788:             )
4789:
4790:         for task in asyncio.as_completed(tasks):
4791:             result = await task
4792:             results.append(result)
4793:
4794:         return results
4795:
4796:     async def aggregate_batch_results(
4797:         self, batch_results: Iterable[BatchResult]
4798:     ) -> AggregatedBatchResults:
4799:         """Aggregate results from multiple batches.
4800:
4801:         Args:
4802:             batch_results: Iterable of BatchResults to aggregate
4803:
4804:         Returns:
4805:             AggregatedBatchResults with summary statistics
4806:         """
4807:         start_time = time.perf_counter()
4808:
4809:         total_batches = 0
4810:         total_items = 0
4811:         successful_items = 0
4812:         failed_items = 0
4813:         all_results: list[Any] = []
4814:         all_errors: list[dict[str, Any]] = []
4815:         batch_metrics_list: list[BatchMetrics] = []
4816:
```

```
        for batch_result in batch_results:
            total_batches += 1
            total_items += len(batch_result.items)
            successful_items += batch_result.metrics["successful_items"]
            failed_items += batch_result.metrics["failed_items"]

            for item, result in zip(
                batch_result.items, batch_result.results, strict=False
            ):
                if result is not None:
                    all_results.append(result)

            all_errors.extend(batch_result.errors)
            batch_metrics_list.append(batch_result.metrics)

        end_time = time.perf_counter()
        execution_time_ms = (end_time - start_time) * 1000.0

        aggregated = AggregatedBatchResults(
            total_batches=total_batches,
            total_items=total_items,
            successful_items=successful_items,
            failed_items=failed_items,
            results=all_results,
            errors=all_errors,
            execution_time_ms=execution_time_ms,
            batch_metrics=batch_metrics_list,
        )

        if self.config.enable_instrumentation:
            logger.info(
                f"Aggregated {total_batches} batches: {successful_items}/{total_items} successful "
                f"(success_rate={aggregated.success_rate():.2%}, "
                f"aggregation_time={execution_time_ms:.2f}ms)"
            )

        return aggregated

    async def stream_aggregate_batches(
        self,
        batch_stream: AsyncIterator[BatchResult],
        aggregation_fn: Callable[[Any, Any], Any],
    ) -> Any:
        """Stream aggregation of batch results without accumulating in memory.

        Args:
            batch_stream: Async iterator of BatchResults
            aggregation_fn: Function to aggregate results incrementally
                (accumulator, new_result) -> accumulator

        Returns:
            Final aggregated result

        Example:
            >>> def aggregate_fn(acc, result):
            ...     acc['count'] += 1
```

```
            ...     acc['total'] += result['score']
            ...     return acc
            >>>
            >>> batches = executor.execute_batches(entities, D1Q1_Executor, doc, ctx)
            >>> final = await executor.stream_aggregate_batches(batches, aggregate_fn)
        """
        accumulator = None

        async for batch_result in batch_stream:
            for result in batch_result.results:
                if result is not None:
                    if accumulator is None:
                        accumulator = result
                    else:
                        accumulator = aggregation_fn(accumulator, result)

        return accumulator


================================================================================
FILE: src/farfan_pipeline/core/orchestrator/calibration_context.py
================================================================================

"""
calibration_context.py - Calibration Context Data Structure

This module defines the CalibrationContext dataclass that encapsulates all
contextual information needed for method calibration. It serves as the
canonical context container for the calibration system.

Architectural Constraints:
- [Constraint 1] Factory method validates question IDs against canonical catalog
- Provides stable hash generation for caching layer (Component 4)
"""

from __future__ import annotations

import hashlib
import re
from dataclasses import dataclass, field
from typing import Optional


@dataclass(frozen=True)
class CalibrationContext:
    """
    Immutable context for method calibration.

    Fields:
        question_id: Question identifier (e.g., "D3-Q2")
        dimension: Dimension number (1-6)
        policy_area: Policy area identifier
        unit_quality: Unit-of-analysis quality score [0,1]
        method_position: Position of method in execution chain (1-indexed)
        total_methods: Total number of methods in execution chain
```

```
        question_num: Question number within dimension (1-5)

    Example:
        >>> ctx = CalibrationContext.from_question_id("D3-Q2", unit_quality=0.85)
        >>> ctx.dimension
        3
        >>> ctx.question_num
        2
    """

    question_id: str
    dimension: int
    policy_area: Optional[str] = None
    unit_quality: float = 1.0
    method_position: int = 1
    total_methods: int = 1
    question_num: int = field(init=False)

    def __post_init__(self):
        """Validate and extract question_num from question_id."""
        # Extract question number from question_id
        match = re.match(r"D(\d+)-Q(\d+)", self.question_id)
        if not match:
            raise ValueError(
                f"Invalid question_id format: {self.question_id}. "
                f"Expected format: D{{n}}-Q{{m}} (e.g., 'D3-Q2')"
            )

        dimension_from_id = int(match.group(1))
        question_num = int(match.group(2))

        # Validate dimension consistency
        if dimension_from_id != self.dimension:
            raise ValueError(
                f"Dimension mismatch: question_id has D{dimension_from_id} "
                f"but dimension={self.dimension}"
            )

        # Validate ranges
        if not (1 <= self.dimension <= 6):
            raise ValueError(f"Dimension must be 1-6, got {self.dimension}")

        if not (1 <= question_num <= 5):
            raise ValueError(f"Question number must be 1-5, got {question_num}")

        if not (0 <= self.unit_quality <= 1):
            raise ValueError(
                f"unit_quality must be in [0,1], got {self.unit_quality}"
            )

        if self.method_position < 1 or self.method_position > self.total_methods:
            raise ValueError(
                f"method_position ({self.method_position}) must be in "
                f"[1, {self.total_methods}]"
            )

```

```
4985:         # Use object.__setattr__ since dataclass is frozen
4986:         object.__setattr__(self, 'question_num', question_num)
4987:
4988:     @classmethod
4989:     def from_question_id(
4990:         cls,
4991:         question_id: str,
4992:         policy_area: Optional[str] = None,
4993:         unit_quality: float = 1.0,
4994:         method_position: int = 1,
4995:         total_methods: int = 1,
4996:     ) -> CalibrationContext:
4997:         """
4998:         Factory method to create context from question ID.
4999:
5000:         Automatically extracts dimension and question numbers from ID format.
5001:
5002:         Args:
5003:             question_id: Question identifier (e.g., "D3-Q2")
5004:             policy_area: Optional policy area identifier
5005:             unit_quality: Unit-of-analysis quality score [0,1]
5006:             method_position: Position in execution chain (1-indexed)
5007:             total_methods: Total methods in execution chain
5008:
5009:         Returns:
5010:             CalibrationContext instance
5011:
5012:         Raises:
5013:             ValueError: If question_id format is invalid
5014:
5015:         Example:
5016:             >>> ctx = CalibrationContext.from_question_id(
5017:             ...     "D3-Q2",
5018:             ...     policy_area="education",
5019:             ...     unit_quality=0.85
5020:             ... )
5021:         """
5022:         # Extract dimension from question_id
5023:         match = re.match(r"D(\d+)-Q(\d+)", question_id)
5024:         if not match:
5025:             raise ValueError(
5026:                 f"Invalid question_id format: {question_id}. "
5027:                 f"Expected format: D{{n}}-Q{{m}} (e.g., 'D3-Q2')"
5028:             )
5029:
5030:         dimension = int(match.group(1))
5031:
5032:         return cls(
5033:             question_id=question_id,
5034:             dimension=dimension,
5035:             policy_area=policy_area,
5036:             unit_quality=unit_quality,
5037:             method_position=method_position,
5038:             total_methods=total_methods,
5039:         )
5040:
```

```
    def to_stable_hash(self) -> str:
        """
        Generate stable hash for caching (Component 4).

        Uses all fields except transient metadata (method_position, total_methods)
        to create a deterministic cache key.

        Returns:
            SHA256 hex digest

        Example:
            >>> ctx1 = CalibrationContext.from_question_id("D3-Q2", unit_quality=0.85)
            >>> ctx2 = CalibrationContext.from_question_id("D3-Q2", unit_quality=0.85)
            >>> ctx1.to_stable_hash() == ctx2.to_stable_hash()
            True
        """
        # Only hash fields that affect calibration semantics
        # Exclude method_position and total_methods (execution metadata)
        hash_input = (
            f"qid={self.question_id}|"
            f"dim={self.dimension}|"
            f"policy={self.policy_area}|"
            f"unit_q={self.unit_quality:.6f}|"
            f"q_num={self.question_num}"
        )

        return hashlib.sha256(hash_input.encode('utf-8')).hexdigest()

    def __repr__(self) -> str:
        """String representation for debugging."""
        return (
            f"CalibrationContext("
            f"question_id='{self.question_id}', "
            f"dimension={self.dimension}, "
            f"question_num={self.question_num}, "
            f"unit_quality={self.unit_quality:.3f})"
        )



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/calibration_types.py
================================================================================

"""Calibration Types Module.

Defines core data structures for method calibration including intrinsic scores
and layer-based runtime evaluation.
"""

from __future__ import annotations

from dataclasses import dataclass


@dataclass(frozen=True)
```

```
5097: class MethodCalibration:
5098:     """Calibration parameters for a method.
5099:
5100:     Attributes:
5101:         score_min: Minimum acceptable score
5102:         score_max: Maximum acceptable score
5103:         min_evidence_snippets: Minimum evidence snippets required
5104:         max_evidence_snippets: Maximum evidence snippets to consider
5105:         contradiction_tolerance: Tolerance for contradictions (0.0-1.0)
5106:         uncertainty_penalty: Penalty for uncertainty (0.0-1.0)
5107:         aggregation_weight: Weight in aggregation
5108:         sensitivity: Sensitivity parameter (0.0-1.0)
5109:         requires_numeric_support: Whether numeric support is required
5110:         requires_temporal_support: Whether temporal support is required
5111:         requires_source_provenance: Whether source provenance is required
5112:     """
5113:     score_min: float = 0.0
5114:     score_max: float = 1.0
5115:     min_evidence_snippets: int = 1
5116:     max_evidence_snippets: int = 10
5117:     contradiction_tolerance: float = 0.3
5118:     uncertainty_penalty: float = 0.5
5119:     aggregation_weight: float = 1.0
5120:     sensitivity: float = 0.5
5121:     requires_numeric_support: bool = False
5122:     requires_temporal_support: bool = False
5123:     requires_source_provenance: bool = True
5124:
5125:
5126: @dataclass(frozen=True)
5127: class IntrinsicScores:
5128:     """Intrinsic calibration scores (@b) for a method.
5129:
5130:     Attributes:
5131:         b_theory: Theoretical soundness score (0.0-1.0)
5132:         b_impl: Implementation quality score (0.0-1.0)
5133:         b_deploy: Deployment readiness score (0.0-1.0)
5134:     """
5135:     b_theory: float
5136:     b_impl: float
5137:     b_deploy: float
5138:
5139:     def average(self) -> float:
5140:         """Compute average intrinsic score."""
5141:         return (self.b_theory + self.b_impl + self.b_deploy) / 3.0
5142:
5143:
5144: @dataclass(frozen=True)
5145: class RuntimeLayers:
5146:     """Runtime calibration layers for dynamic evaluation.
5147:
5148:     Attributes:
5149:         chain: Chain of evidence score (@chain)
5150:         quality: Quality of data/evidence (@q)
5151:         density: Data density score (@d)
5152:         provenance: Provenance traceability (@p)
```

```
        coverage: Coverage completeness (@C)
        uncertainty: Uncertainty quantification (@u)
        mechanism: Mechanistic explanation (@m)
    """
    chain: float = 0.0
    quality: float = 0.0
    density: float = 0.0
    provenance: float = 0.0
    coverage: float = 0.0
    uncertainty: float = 0.0
    mechanism: float = 0.0

    def to_dict(self) -> dict[str, float]:
        """Convert to dictionary for logging."""
        return {
            'chain': self.chain,
            'quality': self.quality,
            'density': self.density,
            'provenance': self.provenance,
            'coverage': self.coverage,
            'uncertainty': self.uncertainty,
            'mechanism': self.mechanism
        }


@dataclass
class LayerRequirements:
    """Layer requirements from canonical inventory.

    Attributes:
        required_layers: List of required layer names
        weights: Dictionary mapping layer names to weights
        aggregation_method: 'choquet_integral' or 'weighted_sum'
    """
    required_layers: list[str]
    weights: dict[str, float]
    aggregation_method: str = 'weighted_sum'

    def validate(self) -> None:
        """Validate layer requirements."""
        if not self.required_layers:
            raise ValueError("No required layers specified")
        if self.aggregation_method not in ('choquet_integral', 'weighted_sum'):
            raise ValueError(f"Invalid aggregation method: {self.aggregation_method}")
        for layer in self.required_layers:
            if layer not in self.weights:
                raise ValueError(f"Missing weight for required layer: {layer}")


__all__ = [
    'MethodCalibration',
    'IntrinsicScores',
    'RuntimeLayers',
    'LayerRequirements'
]

```

```
5209:
5210:
5211: ================================================================================
5212: FILE: src/farfan_pipeline/core/orchestrator/chunk_matrix_builder.py
5213: ================================================================================
5214:
5215: """Chunk matrix construction and validation for policy analysis pipeline.
5216:
5217: This module provides deterministic construction of the 60-chunk PAÃ\227DIM matrix
5218: with comprehensive validation, duplicate detection, and audit logging.
5219:
5220: Validation Hierarchy (Leaf Node Pattern):
5221: ==========================================
5222: Each validation function is atomic with a single failure mode. This design ensures:
5223: - Precise error messages pinpointing the exact validation failure
5224: - Easy debugging by mapping error message to specific validation function
5225: - No cascading failures from complex validation logic
5226:
5227: Validation Stages:
5228: 1. Structure Validation (_validate_document_structure)
5229:    - Document has chunks attribute
5230:    - Chunks is a non-empty list
5231:    - Processing mode is 'chunked'
5232:
5233: 2. Chunk Type Validation (_validate_chunk_structure)
5234:    - Each chunk is a ChunkData instance
5235:
5236: 3. Required Fields Validation (_validate_chunk_required_fields)
5237:    - text, policy_area_id, dimension_id are present and non-null
5238:
5239: 4. Field Type Validation (_validate_chunk_field_types)
5240:    - Fields have correct types (strings)
5241:    - Text content is non-empty
5242:
5243: 5. Format Validation (_validate_chunk_id_format)
5244:    - chunk_id matches PA{01-10}-DIM{01-06} pattern
5245:    - PA value is 01-10
5246:    - DIM value is 01-06
5247:
5248: 6. Consistency Validation (_validate_chunk_id_consistency)
5249:    - chunk_id matches policy_area_id-dimension_id
5250:
5251: 7. Uniqueness Validation (_check_duplicate_key, _check_duplicate_chunk_id)
5252:    - No duplicate PAÃ\227DIM combinations
5253:    - No duplicate chunk_id values
5254:
5255: 8. Completeness Validation (_validate_completeness)
5256:    - All 60 PAÃ\227DIM combinations present
5257:    - No missing policy areas or dimensions
5258:
5259: 9. Cardinality Validation (_validate_chunk_count)
5260:    - Exactly 60 chunks in matrix
5261:
5262: Phase 1 Contract Guarantees:
5263: ============================
5264: - Exactly 60 chunks (10 PA Ã\227 6 DIM)
```

```
- All chunks have valid chunk_id (PA{01-10}-DIM{01-06})
- All chunks have non-empty text content
- No duplicate PAÃ\227DIM combinations
- Complete coverage of all PAÃ\227DIM combinations
- Immutable ChunkData instances (frozen dataclass)
- Efficient O(1) lookup by (PA, DIM) key

Error Message Specificity:
==========================
All error messages follow this pattern:
- Clear identification of what failed
- Location in document (chunk index, PAÃ\227DIM combination)
- Expected vs actual values
- Guidance on which Phase 1 subphase to check
- No generic "validation failed" messages

Example Usage:
==============
    from farfan_pipeline.core.orchestrator.chunk_matrix_builder import (
        build_chunk_matrix,
        validate_chunk_matrix_contract,
        generate_validation_summary,
    )

    # Build and validate in one call (raises on error)
    matrix, keys = build_chunk_matrix(preprocessed_doc)

    # Or validate separately for reporting (no exception)
    report = validate_chunk_matrix_contract(matrix, keys)
    if not report['passed']:
        print(generate_validation_summary(report))
"""

import logging
import re
from functools import lru_cache
from collections.abc import Iterable
from typing import Dict, List, Set, Tuple

from farfan_pipeline.core.types import ChunkData, PreprocessedDocument
from farfan_pipeline.core.orchestrator.questionnaire import load_questionnaire

logger = logging.getLogger(__name__)

CHUNK_ID_PATTERN = re.compile(r"^PA(0[1-9]|10)-DIM(0[1-6])$")
MAX_MISSING_KEYS_TO_DISPLAY = 10


class ChunkMatrixValidationError(ValueError):
    """Raised when chunk matrix validation fails in Phase 1.

    This exception indicates that Phase 1 output does not meet the required
    contract for chunk matrix construction. All errors include specific
    diagnostic information for debugging Phase 1 issues.
    """

```

```
    def __init__(
        self,
        message: str,
        validation_type: str = "unknown",
        details: Dict = None,
        failed_chunk_indices: List[int] = None,
        phase1_subphase: str = None,
    ):
        """Initialize validation error with diagnostic information.

        Args:
            message: Human-readable error message
            validation_type: Category of validation failure
                (e.g., 'structure', 'format', 'completeness')
            details: Additional diagnostic information
            failed_chunk_indices: Indices of chunks that failed validation
            phase1_subphase: Phase 1 subphase where error likely originated
                (SP0-SP15)
        """
        super().__init__(message)
        self.validation_type = validation_type
        self.details = details or {}
        self.failed_chunk_indices = failed_chunk_indices or []
        self.phase1_subphase = phase1_subphase

    def get_diagnostic_report(self) -> str:
        """Generate a comprehensive diagnostic report for debugging.

        Returns:
            Multi-line string with full diagnostic information
        """
        lines = [
            "=" * 80,
            "PHASE 1 CHUNK MATRIX VALIDATION ERROR",
            "=" * 80,
            f"Validation Type: {self.validation_type}",
            f"Error Message: {self}",
        ]

        if self.phase1_subphase:
            lines.append(f"Likely Origin: Phase 1 {self.phase1_subphase}")

        if self.failed_chunk_indices:
            lines.append(f"Failed Chunk Indices: {self.failed_chunk_indices[:20]}")
            if len(self.failed_chunk_indices) > 20:
                lines.append(f"  ... and {len(self.failed_chunk_indices) - 20} more")

        if self.details:
            lines.append("\nDiagnostic Details:")
            for key, value in self.details.items():
                lines.append(f"  {key}: {value}")

        lines.append("=" * 80)
        return "\n".join(lines)
```

```
@lru_cache
def _expected_axes_from_monolith() -> tuple[list[str], list[str]]:
    """Load unique policy areas and dimensions from questionnaire monolith."""
    questionnaire = load_questionnaire()
    micro_questions = questionnaire.get_micro_questions()

    policy_areas = sorted(
        {
            q.get("policy_area_id")
            for q in micro_questions
            if q.get("policy_area_id")
        }
    )
    dimensions = sorted(
        {
            q.get("dimension_id")
            for q in micro_questions
            if q.get("dimension_id")
        }
    )

    if not policy_areas or not dimensions:
        raise ValueError(
            "Questionnaire monolith is missing policy_area_id or dimension_id values"
        )

    return policy_areas, dimensions


POLICY_AREAS, DIMENSIONS = _expected_axes_from_monolith()
EXPECTED_CHUNK_COUNT = len(POLICY_AREAS) * len(DIMENSIONS)


def validate_chunk_matrix_contract(
    matrix: Dict[Tuple[str, str], ChunkData],
    sorted_keys: List[Tuple[str, str]],
) -> Dict[str, any]:
    """Validate that chunk matrix satisfies all Phase 1 output contracts.

    Performs comprehensive validation to ensure Phase 1 produced a valid
    chunk matrix for use in downstream phases (especially Phase 3 routing).

    This function can be called after build_chunk_matrix() to get a detailed
    validation report without raising exceptions.

    Args:
        matrix: Chunk matrix to validate
        sorted_keys: Sorted list of matrix keys

    Returns:
        Validation report dict with:
            - passed (bool): True if all validations passed
            - errors (list): List of error messages
            - warnings (list): List of warning messages
            - metrics (dict): Quantitative metrics about the matrix
            - recommendations (list): Suggestions for fixing issues
```

```
    """
    report = {
        "passed": True,
        "errors": [],
        "warnings": [],
        "recommendations": [],
        "metrics": {
            "chunk_count": len(matrix),
            "expected_count": EXPECTED_CHUNK_COUNT,
            "unique_pa_count": len(set(k[0] for k in sorted_keys)),
            "unique_dim_count": len(set(k[1] for k in sorted_keys)),
            "empty_chunks": 0,
            "chunks_with_provenance": 0,
            "avg_text_length": 0,
        },
    }

    text_lengths = []

    if len(matrix) != EXPECTED_CHUNK_COUNT:
        report["passed"] = False
        deficit = EXPECTED_CHUNK_COUNT - len(matrix)
        if len(matrix) < EXPECTED_CHUNK_COUNT:
            report["errors"].append(
                f"Chunk count deficit: expected {EXPECTED_CHUNK_COUNT}, "
                f"got {len(matrix)} (missing {deficit})"
            )
            report["recommendations"].append(
                "Check Phase 1 segmentation (SP4) to ensure all PAÃ\227DIM combinations are generated"
            )
        else:
            report["errors"].append(
                f"Chunk count surplus: expected {EXPECTED_CHUNK_COUNT}, "
                f"got {len(matrix)} (extra {-deficit})"
            )
            report["recommendations"].append(
                "Check Phase 1 deduplication (SP14) to ensure no duplicate PAÃ\227DIM combinations"
            )

    for (pa, dim), chunk in matrix.items():
        if chunk.policy_area_id != pa:
            report["passed"] = False
            report["errors"].append(
                f"Chunk ({pa}, {dim}) has inconsistent policy_area_id: {chunk.policy_area_id}"
            )
            report["recommendations"].append(
                f"Verify Phase 1 PAÃ\227DIM assignment for chunk at position ({pa}, {dim})"
            )

        if chunk.dimension_id != dim:
            report["passed"] = False
            report["errors"].append(
                f"Chunk ({pa}, {dim}) has inconsistent dimension_id: {chunk.dimension_id}"
            )
            report["recommendations"].append(
                f"Verify Phase 1 PAÃ\227DIM assignment for chunk at position ({pa}, {dim})"
```

```
            )

        if not chunk.text or not chunk.text.strip():
            report["passed"] = False
            report["errors"].append(
                f"Chunk ({pa}, {dim}) has empty text content"
            )
            report["metrics"]["empty_chunks"] += 1
            report["recommendations"].append(
                "Check Phase 1 text extraction and chunking logic "
                "to ensure all chunks receive content"
            )
        else:
            text_lengths.append(len(chunk.text))

        if chunk.provenance is not None:
            report["metrics"]["chunks_with_provenance"] += 1

        expected_chunk_id = f"{pa}-{dim}"
        if chunk.chunk_id and chunk.chunk_id != expected_chunk_id:
            report["warnings"].append(
                f"Chunk ({pa}, {dim}) has unexpected chunk_id: "
                f"{chunk.chunk_id} (expected {expected_chunk_id})"
            )

    if text_lengths:
        report["metrics"]["avg_text_length"] = sum(text_lengths) // len(text_lengths)
        report["metrics"]["min_text_length"] = min(text_lengths)
        report["metrics"]["max_text_length"] = max(text_lengths)

    expected_keys = {(pa, dim) for pa in POLICY_AREAS for dim in DIMENSIONS}
    actual_keys = set(matrix.keys())

    missing_keys = expected_keys - actual_keys
    if missing_keys:
        report["passed"] = False
        report["errors"].append(
            f"Missing PAÃ\227DIM combinations ({len(missing_keys)}): {sorted(missing_keys)[:10]}"
        )
        report["recommendations"].append(
            "Review Phase 1 subphase SP4 (segmentation) output "
            "to identify why some PAÃ\227DIM cells are missing"
        )

    extra_keys = actual_keys - expected_keys
    if extra_keys:
        report["passed"] = False
        report["errors"].append(
            f"Unexpected PAÃ\227DIM combinations: {sorted(extra_keys)}"
        )
        report["recommendations"].append(
            "Verify that POLICY_AREAS and DIMENSIONS constants match Phase 1 configuration"
        )

    if not report["errors"]:
        report["metrics"]["validation_status"] = "PASSED"
```

```
    else:
        report["metrics"]["validation_status"] = "FAILED"

    provenance_rate = report["metrics"]["chunks_with_provenance"] / len(matrix) if matrix else 0.0
    report["metrics"]["provenance_completeness"] = round(provenance_rate, 3)

    if provenance_rate < 0.8:
        report["warnings"].append(
            f"Low provenance completeness: {provenance_rate:.1%} of chunks have provenance data"
        )

    return report


def generate_validation_summary(report: Dict[str, any]) -> str:
    """Generate a human-readable validation summary from a validation report.

    Args:
        report: Validation report from validate_chunk_matrix_contract()

    Returns:
        Formatted multi-line string suitable for logging or display
    """
    lines = [
        "=" * 80,
        "PHASE 1 CHUNK MATRIX VALIDATION SUMMARY",
        "=" * 80,
        f"Status: {'PASSED â\234\223' if report['passed'] else 'FAILED â\234\227'}",
        "",
        "Metrics:",
        f"  Total Chunks: {report['metrics']['chunk_count']} / "
        f"{report['metrics']['expected_count']}",
        f"  Unique Policy Areas: {report['metrics']['unique_pa_count']}",
        f"  Unique Dimensions: {report['metrics']['unique_dim_count']}",
    ]

    if "avg_text_length" in report["metrics"] and report["metrics"]["avg_text_length"] > 0:
        lines.extend([
            f"  Avg Text Length: {report['metrics']['avg_text_length']} chars",
            f"  Text Length Range: {report['metrics'].get('min_text_length', 0)} - "
            f"{report['metrics'].get('max_text_length', 0)} chars",
        ])

    if "provenance_completeness" in report["metrics"]:
        lines.append(
            f"  Provenance Completeness: {report['metrics']['provenance_completeness']:.1%}"
        )

    if report["errors"]:
        lines.extend([
            "",
            f"Errors ({len(report['errors'])}):",
        ])
        for error in report["errors"][:10]:
            lines.append(f"  â\234\227 {error}")
        if len(report["errors"]) > 10:
```

```
            lines.append(f"  ... and {len(report['errors']) - 10} more errors")

    if report["warnings"]:
        lines.extend([
            "",
            f"Warnings ({len(report['warnings'])}):",
        ])
        for warning in report["warnings"][:10]:
            lines.append(f"  â\232  {warning}")
        if len(report["warnings"]) > 10:
            lines.append(f"  ... and {len(report['warnings']) - 10} more warnings")

    if report["recommendations"]:
        lines.extend([
            "",
            "Recommendations:",
        ])
        for rec in set(report["recommendations"][:5]):
            lines.append(f"  â\206\222 {rec}")

    lines.append("=" * 80)
    return "\n".join(lines)


def build_chunk_matrix(
    document: PreprocessedDocument,
) -> tuple[dict[tuple[str, str], ChunkData], list[tuple[str, str]]]:
    """Construct validated chunk matrix from preprocessed document.

    Builds a dictionary mapping (PA, DIM) tuples to ChunkData instances,
    performs comprehensive validation, and returns sorted keys for deterministic
    iteration. Guarantees immutability through frozen ChunkData dataclass.

    Args:
        document: PreprocessedDocument containing 60 policy chunks

    Returns:
        Tuple of (chunk_matrix, sorted_keys) where:
        - chunk_matrix: dict mapping (PA, DIM) -> ChunkData (immutable)
        - sorted_keys: list of (PA, DIM) tuples sorted deterministically

    Raises:
        ValueError: If validation fails with specific error message indicating:
            - Missing or malformed chunk data
            - Missing required fields (chunk_id, policy_area_id, dimension_id, text)
            - Invalid chunk_id format (not PA{01-10}-DIM{01-06})
            - Wrong chunk count (not 60)
            - Duplicate PAÃ\227DIM combinations
            - Missing PAÃ\227DIM combinations
            - Null or empty required fields

    Example:
        >>> doc = PreprocessedDocument(...)
        >>> matrix, keys = build_chunk_matrix(doc)
        >>> chunk = matrix[("PA01", "DIM01")]
        >>> assert len(keys) == 60
```

```
5657:         >>> assert keys[0] == ("PA01", "DIM01")
5658:         >>> # Matrix is immutable - ChunkData is frozen
5659:     """
5660:     logger.info(
5661:         f"Phase 1 Chunk Matrix Construction: document={document.document_id}, "
5662:         f"input_chunk_count={len(document.chunks)}"
5663:     )
5664:
5665:     _validate_document_structure(document)
5666:
5667:     matrix: Dict[Tuple[str, str], ChunkData] = {}
5668:     seen_keys: Set[Tuple[str, str]] = set()
5669:     seen_chunk_ids: Set[str] = set()
5670:     validation_errors: List[str] = []
5671:
5672:     for idx, chunk in enumerate(document.chunks):
5673:         try:
5674:             _validate_chunk_structure(chunk, idx)
5675:             _validate_chunk_required_fields(chunk, idx)
5676:             _validate_chunk_field_types(chunk, idx)
5677:
5678:             chunk_id = chunk.chunk_id or f"{chunk.policy_area_id}-{chunk.dimension_id}"
5679:             _validate_chunk_id_format(chunk_id, idx)
5680:
5681:             key = (chunk.policy_area_id, chunk.dimension_id)
5682:             _validate_chunk_id_consistency(chunk_id, key, idx)
5683:
5684:             _check_duplicate_key(key, seen_keys, chunk_id, idx)
5685:             _check_duplicate_chunk_id(chunk_id, seen_chunk_ids, idx)
5686:
5687:             seen_keys.add(key)
5688:             seen_chunk_ids.add(chunk_id)
5689:             matrix[key] = chunk
5690:
5691:         except ValueError as e:
5692:             validation_errors.append(str(e))
5693:             logger.error(f"Chunk validation failed at index {idx}: {e}")
5694:
5695:     if validation_errors:
5696:         error_summary = "\n  - ".join(validation_errors[:10])
5697:         remaining = len(validation_errors) - 10
5698:         if remaining > 0:
5699:             error_summary += f"\n  ... and {remaining} more errors"
5700:         raise ValueError(
5701:             f"Phase 1 chunk matrix validation failed with "
5702:             f"{len(validation_errors)} error(s):\n  - {error_summary}"
5703:         )
5704:
5705:     _validate_completeness(seen_keys, POLICY_AREAS, DIMENSIONS)
5706:     _validate_chunk_count(matrix, EXPECTED_CHUNK_COUNT)
5707:
5708:     sorted_keys = _sort_keys_deterministically(matrix.keys())
5709:
5710:     logger.info(
5711:         f"Phase 1 chunk matrix constructed successfully: unique_chunks={len(matrix)}, "
5712:         f"expected={EXPECTED_CHUNK_COUNT}, all_validations_passed=True"
```

```
    )
    _log_audit_summary(matrix, sorted_keys)

    return matrix, sorted_keys


def _validate_document_structure(document: PreprocessedDocument) -> None:
    """Validate document structure before processing chunks.

    Atomic validation: document has chunks list with valid structure.

    Args:
        document: PreprocessedDocument to validate

    Raises:
        ValueError: If document structure is invalid
    """
    if not hasattr(document, 'chunks'):
        raise ValueError(
            "PreprocessedDocument missing 'chunks' attribute. "
            "Ensure Phase 1 SPC ingestion completed successfully. "
            "Check that Phase 1 execution (all 16 subphases) completed without errors."
        )

    if not isinstance(document.chunks, list):
        raise ValueError(
            f"PreprocessedDocument.chunks must be a list, got {type(document.chunks).__name__}. "
            "This indicates corrupted Phase 1 output. "
            "Phase 1 must produce a list of ChunkData instances."
        )

    if len(document.chunks) == 0:
        raise ValueError(
            "PreprocessedDocument.chunks is empty. "
            "Phase 1 must produce exactly 60 chunks (10 PA Ã\227 6 DIM). "
            "Check Phase 1 segmentation (SP4) and smart chunk generation (SP11)."
        )

    if not hasattr(document, 'document_id') or not document.document_id:
        raise ValueError(
            "PreprocessedDocument missing or empty document_id. "
            "Phase 0 input validation must set document_id, and Phase 1 must preserve it."
        )

    if not hasattr(document, 'processing_mode'):
        raise ValueError(
            "PreprocessedDocument missing processing_mode attribute. "
            "Phase 1 must set processing_mode to 'chunked'."
        )

    if document.processing_mode != 'chunked':
        raise ValueError(
            f"PreprocessedDocument.processing_mode must be 'chunked', got '{document.processing_mode}'. "
            "Phase 1 must set processing_mode to 'chunked' for chunk-aware routing."
        )

```

```
5769:
5770: def _validate_chunk_structure(chunk: ChunkData, idx: int) -> None:
5771:     """Validate chunk is a ChunkData instance.
5772:
5773:     Atomic validation: chunk type check.
5774:
5775:     Args:
5776:         chunk: Object to validate
5777:         idx: Chunk index for error reporting
5778:
5779:     Raises:
5780:         ValueError: If chunk is not ChunkData instance
5781:     """
5782:     if not isinstance(chunk, ChunkData):
5783:         raise ValueError(
5784:             f"Chunk at index {idx}: expected ChunkData instance, got {type(chunk).__name__}. "
5785:             "Phase 1 output must contain only ChunkData instances."
5786:         )
5787:
5788:
5789: def _validate_chunk_required_fields(chunk: ChunkData, idx: int) -> None:
5790:     """Validate chunk has all required fields with non-null values.
5791:
5792:     Atomic validation: presence of required fields.
5793:
5794:     Args:
5795:         chunk: ChunkData to validate
5796:         idx: Chunk index for error reporting
5797:
5798:     Raises:
5799:         ValueError: If any required field is missing or None
5800:     """
5801:     required_fields = {
5802:         'text': 'text_content',
5803:         'policy_area_id': 'policy_area_id (PA01-PA10)',
5804:         'dimension_id': 'dimension_id (DIM01-DIM06)',
5805:     }
5806:
5807:     missing_fields = []
5808:     null_fields = []
5809:
5810:     for field, description in required_fields.items():
5811:         if not hasattr(chunk, field):
5812:             missing_fields.append(description)
5813:         elif getattr(chunk, field) is None:
5814:             null_fields.append(description)
5815:
5816:     if missing_fields:
5817:         raise ValueError(
5818:             f"Chunk at index {idx} (id={chunk.id}): missing required field(s) {', '.join(missing_fields)}. "
5819:             "All chunks must have text, policy_area_id, and dimension_id."
5820:         )
5821:
5822:     if null_fields:
5823:         raise ValueError(
5824:             f"Chunk at index {idx} (id={chunk.id}): null value in required field(s) {', '.join(null_fields)}. "
```

```
5825:             "Phase 1 must populate all required fields with non-null values."
5826:         )
5827:
5828:
5829: def _validate_chunk_field_types(chunk: ChunkData, idx: int) -> None:
5830:     """Validate chunk field types and non-empty values.
5831:
5832:     Atomic validation: field type and content checks.
5833:
5834:     Args:
5835:         chunk: ChunkData to validate
5836:         idx: Chunk index for error reporting
5837:
5838:     Raises:
5839:         ValueError: If field types are invalid or values are empty
5840:     """
5841:     if not isinstance(chunk.text, str):
5842:         raise ValueError(
5843:             f"Chunk at index {idx} (id={chunk.id}): text must be string, got {type(chunk.text).__name__}"
5844:         )
5845:
5846:     if not chunk.text.strip():
5847:         raise ValueError(
5848:             f"Chunk at index {idx} (id={chunk.id}): text_content is empty or whitespace-only. "
5849:             "All chunks must contain non-empty text."
5850:         )
5851:
5852:     if not isinstance(chunk.policy_area_id, str):
5853:         raise ValueError(
5854:             f"Chunk at index {idx} (id={chunk.id}): policy_area_id must be string, got {type(chunk.policy_area_id).__name__}"
5855:         )
5856:
5857:     if not isinstance(chunk.dimension_id, str):
5858:         raise ValueError(
5859:             f"Chunk at index {idx} (id={chunk.id}): dimension_id must be string, got {type(chunk.dimension_id).__name__}"
5860:         )
5861:
5862:
5863: def _validate_chunk_id_format(chunk_id: str, idx: int) -> None:
5864:     """Validate chunk_id matches PA{01-10}-DIM{01-06} pattern.
5865:
5866:     Atomic validation: chunk_id format compliance.
5867:
5868:     Args:
5869:         chunk_id: Chunk identifier to validate
5870:         idx: Chunk index for error reporting
5871:
5872:     Raises:
5873:         ValueError: If chunk_id format is invalid with specific pattern expected
5874:     """
5875:     if not isinstance(chunk_id, str):
5876:         raise ValueError(
5877:             f"Chunk at index {idx}: chunk_id must be string, got {type(chunk_id).__name__}"
5878:         )
5879:
5880:     if not chunk_id.strip():
```

```
5881:         raise ValueError(
5882:             f"Chunk at index {idx}: chunk_id is empty or whitespace-only"
5883:         )
5884:
5885:     if not CHUNK_ID_PATTERN.match(chunk_id):
5886:         match = re.match(r'^(PA\d{2})-(DIM\d{2})$', chunk_id)
5887:         if match:
5888:             pa_part, dim_part = match.groups()
5889:
5890:             pa_match = re.match(r'^PA(\d{2})$', pa_part)
5891:             if pa_match:
5892:                 pa_num = int(pa_match.group(1))
5893:                 if pa_num < 1 or pa_num > 10:
5894:                     raise ValueError(
5895:                         f"Chunk at index {idx}: invalid chunk_id '{chunk_id}'. "
5896:                         f"Policy area must be PA01-PA10, got {pa_part} (value {pa_num} out of range)"
5897:                     )
5898:
5899:             dim_match = re.match(r'^DIM(\d{2})$', dim_part)
5900:             if dim_match:
5901:                 dim_num = int(dim_match.group(1))
5902:                 if dim_num < 1 or dim_num > 6:
5903:                     raise ValueError(
5904:                         f"Chunk at index {idx}: invalid chunk_id '{chunk_id}'. "
5905:                         f"Dimension must be DIM01-DIM06, got {dim_part} (value {dim_num} out of range)"
5906:                     )
5907:
5908:         raise ValueError(
5909:             f"Chunk at index {idx}: invalid chunk_id format '{chunk_id}'. "
5910:             "Expected format: PA{{01-10}}-DIM{{01-06}} (e.g., 'PA01-DIM01', 'PA10-DIM06')"
5911:         )
5912:
5913:
5914: def _validate_chunk_id_consistency(
5915:     chunk_id: str, key: Tuple[str, str], idx: int
5916: ) -> None:
5917:     """Validate chunk_id is consistent with policy_area_id and dimension_id.
5918:
5919:     Atomic validation: chunk_id matches PAÃ\227DIM key.
5920:
5921:     Args:
5922:         chunk_id: Chunk identifier string
5923:         key: Tuple of (policy_area_id, dimension_id)
5924:         idx: Chunk index for error reporting
5925:
5926:     Raises:
5927:         ValueError: If chunk_id doesn't match the PA-DIM format from key
5928:     """
5929:     expected_chunk_id = f"{key[0]}-{key[1]}"
5930:     if chunk_id != expected_chunk_id:
5931:         raise ValueError(
5932:             f"Chunk at index {idx}: chunk_id inconsistency detected. "
5933:             f"chunk_id='{chunk_id}' does not match policy_area_id='{key[0]}' and dimension_id='{key[1]}' "
5934:             f"(expected '{expected_chunk_id}'). "
5935:             "Phase 1 must ensure chunk_id equals 'policy_area_id-dimension_id'."
5936:         )
```

```
5937:
5938:
5939: def _check_duplicate_key(
5940:     key: Tuple[str, str],
5941:     seen_keys: Set[Tuple[str, str]],
5942:     chunk_id: str,
5943:     idx: int,
5944: ) -> None:
5945:     """Check for duplicate (PA, DIM) keys.
5946:
5947:     Atomic validation: uniqueness of PAÃ\227DIM combination.
5948:
5949:     Args:
5950:         key: (policy_area_id, dimension_id) tuple
5951:         seen_keys: Set of previously seen keys
5952:         chunk_id: Chunk identifier for error reporting
5953:         idx: Chunk index for error reporting
5954:
5955:     Raises:
5956:         ValueError: If key already exists in seen_keys with specific guidance
5957:     """
5958:     if key in seen_keys:
5959:         raise ValueError(
5960:             f"Chunk at index {idx}: duplicate (PA, DIM) combination detected for '{chunk_id}'. "
5961:             f"The combination ({key[0]}, {key[1]}) already exists in the matrix. "
5962:             "Each PAÃ\227DIM combination must appear exactly once. "
5963:             "Phase 1 must produce unique chunks for all 60 combinations (10 PA Ã\227 6 DIM)."
5964:         )
5965:
5966:
5967: def _check_duplicate_chunk_id(
5968:     chunk_id: str,
5969:     seen_chunk_ids: Set[str],
5970:     idx: int,
5971: ) -> None:
5972:     """Check for duplicate chunk_id strings.
5973:
5974:     Atomic validation: uniqueness of chunk_id.
5975:
5976:     Args:
5977:         chunk_id: Chunk identifier to check
5978:         seen_chunk_ids: Set of previously seen chunk IDs
5979:         idx: Chunk index for error reporting
5980:
5981:     Raises:
5982:         ValueError: If chunk_id already exists with guidance on cause
5983:     """
5984:     if chunk_id in seen_chunk_ids:
5985:         raise ValueError(
5986:             f"Chunk at index {idx}: duplicate chunk_id '{chunk_id}'. "
5987:             f"Each chunk must have a unique identifier. "
5988:             "This error typically indicates Phase 1 produced multiple chunks with the same PAÃ\227DIM values, "
5989:             "or chunk_id was manually set to a duplicate value."
5990:         )
5991:
5992:
```

```
def _validate_chunk_count(
    matrix: Dict[Tuple[str, str], ChunkData],
    expected_count: int,
) -> None:
    """Validate chunk matrix has exactly the expected number of chunks.

    Atomic validation: chunk count equals 60.

    Args:
        matrix: Chunk matrix keyed by (PA, DIM)
        expected_count: Expected number of chunks (60)

    Raises:
        ValueError: If chunk count doesn't match with diagnostic information
    """
    actual_count = len(matrix)
    if actual_count != expected_count:
        deficit = expected_count - actual_count
        surplus = actual_count - expected_count

        if actual_count < expected_count:
            raise ValueError(
                f"Phase 1 chunk matrix cardinality violation: expected {expected_count} unique chunks (10 PA Ã\227 6 DIM), "
                f"but found {actual_count} (deficit of {deficit}). "
                "This indicates Phase 1 failed to produce all required PAÃ\227DIM combinations. "
                "Check Phase 1 output for missing policy areas or dimensions."
            )
        else:
            raise ValueError(
                f"Phase 1 chunk matrix cardinality violation: expected {expected_count} unique chunks (10 PA Ã\227 6 DIM), "
                f"but found {actual_count} (surplus of {surplus}). "
                "This indicates Phase 1 produced duplicate PAÃ\227DIM combinations that were not caught. "
                "Verify Phase 1 deduplication logic (SP14)."
            )


def _validate_completeness(
    seen_keys: Set[Tuple[str, str]],
    policy_areas: List[str],
    dimensions: List[str],
) -> None:
    """Validate all required PAÃ\227DIM combinations are present.

    Atomic validation: completeness of PAÃ\227DIM coverage.

    Args:
        seen_keys: Set of (PA, DIM) keys found in document
        policy_areas: List of expected policy areas (PA01-PA10)
        dimensions: List of expected dimensions (DIM01-DIM06)

    Raises:
        ValueError: If any required combinations are missing with detailed diagnostic
    """
    expected_keys = {(pa, dim) for pa in policy_areas for dim in dimensions}
    missing_keys = expected_keys - seen_keys

```

```
    if missing_keys:
        missing_sorted = sorted(missing_keys)
        missing_count = len(missing_keys)

        missing_by_pa: Dict[str, List[str]] = {}
        missing_by_dim: Dict[str, List[str]] = {}

        for pa, dim in missing_sorted:
            missing_by_pa.setdefault(pa, []).append(dim)
            missing_by_dim.setdefault(dim, []).append(pa)

        display_limit = 15
        missing_display = [f"{pa}-{dim}" for pa, dim in missing_sorted[:display_limit]]

        error_parts = [
            f"Phase 1 chunk matrix completeness violation: missing {missing_count} PAÃ\227DIM combination(s).",
            f"Missing combinations: [{', '.join(missing_display)}",
        ]

        if missing_count > display_limit:
            error_parts.append(f", ... and {missing_count - display_limit} more")

        error_parts.append("]")

        if len(missing_by_pa) <= 3:
            pa_summary = "; ".join(
                f"{pa}: missing {len(dims)} dimension(s) ({', '.join(dims)})"
                for pa, dims in sorted(missing_by_pa.items())
            )
            error_parts.append(f"\nBy policy area: {pa_summary}")

        if len(missing_by_dim) <= 3:
            dim_summary = "; ".join(
                f"{dim}: missing {len(pas)} policy area(s) ({', '.join(pas)})"
                for dim, pas in sorted(missing_by_dim.items())
            )
            error_parts.append(f"\nBy dimension: {dim_summary}")

        error_parts.append(
            "\nPhase 1 must produce exactly 60 chunks covering all combinations. "
            "Check Phase 1 segmentation logic (SP4) and ensure all PAÃ\227DIM cells are populated."
        )

        raise ValueError("".join(error_parts))


def _sort_keys_deterministically(
    keys: Iterable[Tuple[str, str]],
) -> List[Tuple[str, str]]:
    """Sort matrix keys deterministically by PA then DIM.

    Ensures O(1) lookup operations benefit from predictable iteration order.

    Args:
        keys: Iterable of (PA, DIM) tuple keys

```

```
    Returns:
        Sorted list of keys for deterministic iteration (PA01-DIM01, PA01-DIM02, ..., PA10-DIM06)
    """
    return sorted(keys, key=lambda k: (k[0], k[1]))


def _log_audit_summary(
    matrix: Dict[Tuple[str, str], ChunkData],
    sorted_keys: List[Tuple[str, str]],
) -> None:
    """Log comprehensive audit summary of constructed chunk matrix.

    Provides diagnostic information for Phase 1 output quality assessment.

    Args:
        matrix: Constructed chunk matrix (immutable ChunkData instances)
        sorted_keys: Sorted list of matrix keys
    """
    pa_counts: Dict[str, int] = {pa: 0 for pa in POLICY_AREAS}
    dim_counts: Dict[str, int] = {dim: 0 for dim in DIMENSIONS}

    text_lengths: List[int] = []
    chunks_with_provenance = 0

    for pa, dim in sorted_keys:
        pa_counts[pa] = pa_counts.get(pa, 0) + 1
        dim_counts[dim] = dim_counts.get(dim, 0) + 1

        chunk = matrix[(pa, dim)]
        text_lengths.append(len(chunk.text))

        if chunk.provenance is not None:
            chunks_with_provenance += 1

    total_text_length = sum(text_lengths)
    avg_text_length = total_text_length // len(matrix) if matrix else 0
    min_text_length = min(text_lengths) if text_lengths else 0
    max_text_length = max(text_lengths) if text_lengths else 0

    provenance_completeness = chunks_with_provenance / len(matrix) if matrix else 0.0

    logger.info(
        "phase1_chunk_matrix_audit_summary",
        extra={
            "total_chunks": len(matrix),
            "expected_chunks": EXPECTED_CHUNK_COUNT,
            "validation_passed": True,
            "chunks_per_policy_area": pa_counts,
            "chunks_per_dimension": dim_counts,
            "text_statistics": {
                "total_chars": total_text_length,
                "avg_length": avg_text_length,
                "min_length": min_text_length,
                "max_length": max_text_length,
            },
            "provenance_completeness": round(provenance_completeness, 3),
```

```
            "chunks_with_provenance": chunks_with_provenance,
            "immutability_guaranteed": True,
        },
    )


__all__ = [
    "build_chunk_matrix",
    "validate_chunk_matrix_contract",
    "generate_validation_summary",
    "ChunkMatrixValidationError",
    "POLICY_AREAS",
    "DIMENSIONS",
    "EXPECTED_CHUNK_COUNT",
    "CHUNK_ID_PATTERN",
]



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/chunk_router.py
================================================================================

"""
Chunk Router for SPC Exploitation.

Routes semantic chunks to appropriate executors based on chunk type,
enabling targeted execution and reducing redundant processing.
"""

from __future__ import annotations

import hashlib
import json
from dataclasses import asdict, dataclass
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from farfan_pipeline.core.types import ChunkData

# Routing table version identifier
ROUTING_TABLE_VERSION = "v1"


@dataclass
class ChunkRoute:
    """Routing decision for a single chunk."""

    chunk_id: int
    chunk_type: str
    executor_class: str
    methods: list[tuple[str, str]]  # [(class_name, method_name), ...]
    skip_reason: str | None = None


class ChunkRouter:
```

```
    """
    Routes chunks to appropriate executors based on semantic type.

    This enables chunk-aware execution, where different chunk types
    are processed by the most relevant executors, avoiding unnecessary
    full-document processing.
    """

    # TYPE-TO-EXECUTOR MAPPING
    # Maps chunk types to executor base slots (e.g., "D1Q1", "D2Q3")
    ROUTING_TABLE: dict[str, list[str]] = {
        "diagnostic": ["D1Q1", "D1Q2", "D1Q5"],  # Baseline/gap analysis executors
        "activity": [
            "D2Q1",
            "D2Q2",
            "D2Q3",
            "D2Q4",
            "D2Q5",
        ],  # Activity/intervention executors
        "indicator": ["D3Q1", "D3Q2", "D4Q1", "D5Q1"],  # Metric/indicator executors
        "resource": ["D1Q3", "D2Q4", "D5Q5"],  # Financial/resource executors
        "temporal": ["D1Q5", "D3Q4", "D5Q4"],  # Timeline/temporal executors
        "entity": ["D2Q3", "D3Q3"],  # Responsibility/entity executors
    }

    # METHODS THAT MUST SEE FULL GRAPH
    # These methods require access to the complete chunk graph
    GRAPH_METHODS: set[str] = {
        "TeoriaCambio.construir_grafo_causal",
        "CausalExtractor.extract_causal_hierarchy",
        "AdvancedDAGValidator.calculate_acyclicity_pvalue",
        "CrossReferenceValidator.validate_internal_consistency",
    }

    def route_chunk(self, chunk: ChunkData) -> ChunkRoute:
        """
        Determine executor routing for a chunk.

        Args:
            chunk: ChunkData to route

        Returns:
            ChunkRoute with executor assignment and method list
        """
        executor_classes = self.ROUTING_TABLE.get(chunk.chunk_type, [])

        if not executor_classes:
            return ChunkRoute(
                chunk_id=chunk.id,
                chunk_type=chunk.chunk_type,
                executor_class="",
                methods=[],
                skip_reason=f"No executor mapping for chunk type '{chunk.chunk_type}'",
            )

        # Get primary executor for this chunk type
```

```
        primary_executor = executor_classes[0]

        # Get method subset for this chunk type
        # Note: Actual method filtering would require loading executor configs
        # For now, we return empty list and let execute_chunk filter
        methods: list[tuple[str, str]] = []

        return ChunkRoute(
            chunk_id=chunk.id,
            chunk_type=chunk.chunk_type,
            executor_class=primary_executor,
            methods=methods,
        )

    def should_use_full_graph(self, method_name: str, class_name: str = "") -> bool:
        """
        Check if a method requires access to the full chunk graph.

        Args:
            method_name: Name of the method
            class_name: Optional class name

        Returns:
            True if method needs full graph access
        """
        full_name = f"{class_name}.{method_name}" if class_name else method_name
        return full_name in self.GRAPH_METHODS or method_name in self.GRAPH_METHODS

    def get_relevant_executors(self, chunk_type: str) -> list[str]:
        """
        Get list of executors relevant to a chunk type.

        Args:
            chunk_type: Type of chunk

        Returns:
            List of executor base slots
        """
        return self.ROUTING_TABLE.get(chunk_type, [])

    def generate_execution_map(self, chunks: list[ChunkData]) -> dict[int, ChunkRoute]:
        """
        Generate a deterministic execution map for a list of chunks.

        This map serves as the binding contract for the Orchestrator,
        dictating exactly which executor processes which chunk.

        Args:
            chunks: List of ChunkData objects

        Returns:
            Dictionary mapping chunk_id to ChunkRoute
        """
        execution_map = {}
        # Sort chunks by ID to ensure deterministic processing order if relevant,
        # though the output dict key order is insertion-ordered in modern Python.
```

```
        # We process them in order to be safe.
        sorted_chunks = sorted(chunks, key=lambda c: c.id)

        for chunk in sorted_chunks:
            route = self.route_chunk(chunk)
            execution_map[chunk.id] = route

        return execution_map


def serialize_execution_map(execution_map: dict[int, ChunkRoute]) -> str:
    """
    Serialize an execution map to JSON string.

    Args:
        execution_map: Dictionary mapping chunk_id to ChunkRoute

    Returns:
        JSON string representation of the execution map
    """
    serializable_map = {
        "version": ROUTING_TABLE_VERSION,
        "routes": {
            str(chunk_id): asdict(route) for chunk_id, route in execution_map.items()
        },
    }
    return json.dumps(serializable_map, sort_keys=True, indent=2)


def deserialize_execution_map(serialized_map: str) -> dict[int, ChunkRoute]:
    """
    Deserialize an execution map from JSON string.

    Args:
        serialized_map: JSON string representation of the execution map

    Returns:
        Dictionary mapping chunk_id to ChunkRoute

    Raises:
        ValueError: If the serialized map is invalid or has wrong version
    """
    try:
        data = json.loads(serialized_map)
    except json.JSONDecodeError as e:
        raise ValueError(f"Invalid JSON format: {e}") from e

    if "version" not in data or data["version"] != ROUTING_TABLE_VERSION:
        raise ValueError(
            f"Invalid or unsupported version: expected {ROUTING_TABLE_VERSION}, "
            f"got {data.get('version', 'missing')}"
        )

    if "routes" not in data:
        raise ValueError("Missing 'routes' key in serialized map")

```

```
    execution_map = {}
    for chunk_id_str, route_dict in data["routes"].items():
        try:
            chunk_id = int(chunk_id_str)
        except ValueError as e:
            raise ValueError(
                f"Invalid chunk_id '{chunk_id_str}': must be an integer"
            ) from e

        valid_fields = {
            "chunk_id",
            "chunk_type",
            "executor_class",
            "methods",
            "skip_reason",
        }
        filtered_dict = {k: v for k, v in route_dict.items() if k in valid_fields}

        try:
            route = ChunkRoute(**filtered_dict)
        except TypeError as e:
            raise ValueError(
                f"Invalid ChunkRoute data for chunk {chunk_id}: {e}"
            ) from e

        execution_map[chunk_id] = route

    return execution_map


def compute_execution_map_hash(execution_map: dict[int, ChunkRoute]) -> str:
    """
    Compute a deterministic hash of an execution map for integrity verification.

    Args:
        execution_map: Dictionary mapping chunk_id to ChunkRoute

    Returns:
        SHA256 hex digest of the serialized map
    """
    serialized = serialize_execution_map(execution_map)
    return hashlib.sha256(serialized.encode("utf-8")).hexdigest()


================================================================================
FILE: src/farfan_pipeline/core/orchestrator/class_registry.py
================================================================================

"""Dynamic class registry for orchestrator method execution."""
from __future__ import annotations

from importlib import import_module
from typing import TYPE_CHECKING

if TYPE_CHECKING:
```

```
    from collections.abc import Mapping

class ClassRegistryError(RuntimeError):
    """Raised when one or more classes cannot be loaded."""

# Map of orchestrator-facing class names to their import paths.
_CLASS_PATHS: Mapping[str, str] = {
    "IndustrialPolicyProcessor": "farfan_core.processing.policy_processor.IndustrialPolicyProcessor",
    "PolicyTextProcessor": "farfan_core.processing.policy_processor.PolicyTextProcessor",
    "BayesianEvidenceScorer": "farfan_core.processing.policy_processor.BayesianEvidenceScorer",
    "PolicyContradictionDetector": "farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector",
    "TemporalLogicVerifier": "farfan_core.analysis.contradiction_deteccion.TemporalLogicVerifier",
    "BayesianConfidenceCalculator": "farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator",
    "PDETMunicipalPlanAnalyzer": "farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer",
    "CDAFFramework": "farfan_core.analysis.derek_beach.CDAFFramework",
    "CausalExtractor": "farfan_core.analysis.derek_beach.CausalExtractor",
    "OperationalizationAuditor": "farfan_core.analysis.derek_beach.OperationalizationAuditor",
    "FinancialAuditor": "farfan_core.analysis.derek_beach.FinancialAuditor",
    "BayesianMechanismInference": "farfan_core.analysis.derek_beach.BayesianMechanismInference",
    "BayesianNumericalAnalyzer": "farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer",
    "PolicyAnalysisEmbedder": "farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder",
    "AdvancedSemanticChunker": "farfan_core.processing.embedding_policy.AdvancedSemanticChunker",
    # SemanticChunker is an alias maintained for backwards compatibility.
    "SemanticChunker": "farfan_core.processing.embedding_policy.AdvancedSemanticChunker",
    "SemanticAnalyzer": "farfan_core.analysis.Analyzer_one.SemanticAnalyzer",
    "PerformanceAnalyzer": "farfan_core.analysis.Analyzer_one.PerformanceAnalyzer",
    "TextMiningEngine": "farfan_core.analysis.Analyzer_one.TextMiningEngine",
    "MunicipalOntology": "farfan_core.analysis.Analyzer_one.MunicipalOntology",
    "TeoriaCambio": "farfan_core.analysis.teoria_cambio.TeoriaCambio",
    "AdvancedDAGValidator": "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator",
    "D1_Q1_QuantitativeBaselineExtractor": "farfan_core.core.orchestrator.executors.D1_Q1_QuantitativeBaselineExtractor",
    "D1_Q2_ProblemDimensioningAnalyzer": "farfan_core.core.orchestrator.executors.D1_Q2_ProblemDimensioningAnalyzer",
    "SemanticProcessor": "farfan_core.processing.semantic_chunking_policy.SemanticProcessor",
    "BayesianCounterfactualAuditor": "farfan_core.analysis.derek_beach.BayesianCounterfactualAuditor",
}

def build_class_registry() -> dict[str, type[object]]:
    """Return a mapping of class names to loaded types, validating availability.

    Classes that depend on optional dependencies (e.g., torch) are skipped
    gracefully if those dependencies are not available.
    """
    resolved: dict[str, type[object]] = {}
    missing: dict[str, str] = {}
    skipped_optional: dict[str, str] = {}

    for name, path in _CLASS_PATHS.items():
        module_name, _, class_name = path.rpartition(".")
        if not module_name:
            missing[name] = path
            continue
        try:
            module = import_module(module_name)
        except ImportError as exc:
            exc_str = str(exc)
            # Check if this is an optional dependency error
```

```
            optional_deps = [
                "torch", "tensorflow", "pyarrow", "camelot",
                "sentence_transformers", "transformers", "spacy",
                "pymc", "arviz", "dowhy", "econml"
            ]
            if any(opt_dep in exc_str for opt_dep in optional_deps):
                # Mark as skipped optional rather than missing
                skipped_optional[name] = f"{path} (optional dependency: {exc})"
            else:
                missing[name] = f"{path} (import error: {exc})"
            continue
        try:
            attr = getattr(module, class_name)
        except AttributeError:
            missing[name] = f"{path} (attribute missing)"
        else:
            if not isinstance(attr, type):
                missing[name] = f"{path} (attribute is not a class: {type(attr).__name__})"
            else:
                resolved[name] = attr

    # Log skipped optional dependencies
    if skipped_optional:
        import logging
        logger = logging.getLogger(__name__)
        logger.info(
            f"Skipped {len(skipped_optional)} optional classes due to missing dependencies: "
            f"{', '.join(skipped_optional.keys())}"
        )

    if missing:
        formatted = ", ".join(f"{name}: {reason}" for name, reason in missing.items())
        raise ClassRegistryError(f"Failed to load orchestrator classes: {formatted}")
    return resolved

def get_class_paths() -> Mapping[str, str]:
    """Expose the raw class path mapping for diagnostics."""
    return _CLASS_PATHS


================================================================================
FILE: src/farfan_pipeline/core/orchestrator/contract_loader.py
================================================================================

"""
Legacy contract loader shim.

This module exists only to satisfy historical imports. The canonical executor
contract loading path is implemented in BaseExecutorWithContract._load_contract.
Use that path instead of this loader for all new code.
"""

from __future__ import annotations

from dataclasses import dataclass
```

```
from typing import Any


class LoadError(RuntimeError):
    """Raised when legacy contract loading is invoked."""


@dataclass
class LoadResult:
    """Placeholder result for legacy contract loading."""

    contracts: dict[str, Any] | None = None
    errors: list[str] | None = None


class JSONContractLoader:
    """Legacy loader shim that fails fast.

    Contract executors should load contracts via BaseExecutorWithContract._load_contract,
    which validates against the canonical schema. This shim prevents silent fallbacks.
    """

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

    def load_file(self, *args: Any, **kwargs: Any) -> LoadResult:
        raise LoadError(
            "JSONContractLoader is obsolete. Use BaseExecutorWithContract._load_contract "
            "for executor contracts."
        )

    def load_directory(self, *args: Any, **kwargs: Any) -> LoadResult:
        raise LoadError(
            "JSONContractLoader is obsolete. Use BaseExecutorWithContract._load_contract "
            "for executor contracts."
        )

    def load_multiple(self, *args: Any, **kwargs: Any) -> LoadResult:
        raise LoadError(
            "JSONContractLoader is obsolete. Use BaseExecutorWithContract._load_contract "
            "for executor contracts."
        )


__all__ = ["JSONContractLoader", "LoadError", "LoadResult"]



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/core.py
================================================================================

"""Clean Orchestrator - VersiÃ³n Completa y Funcional

Orquestador de 11 fases con toda la funcionalidad real preservada.
Sin simplificaciones, sin placeholders.
```

```
6609: """
6610:
6611: from __future__ import annotations
6612:
6613: import asyncio
6614: import inspect
6615: import json
6616: import logging
6617: import os
6618: import statistics
6619: import threading
6620: import time
6621: from collections import deque
6622: from dataclasses import dataclass, field
6623: from datetime import datetime
6624: from pathlib import Path
6625: from types import MappingProxyType
6626: from typing import TYPE_CHECKING, Any, Callable, TypeVar, ParamSpec, TypedDict
6627:
6628: if TYPE_CHECKING:
6629:     from farfan_pipeline.core.orchestrator.factory import CanonicalQuestionnaire
6630:
6631: from farfan_pipeline.core.analysis_port import RecommendationEnginePort
6632: from farfan_pipeline.config.paths import PROJECT_ROOT, RULES_DIR
6633: from farfan_pipeline.processing.aggregation import (
6634:     ClusterScore,
6635:     MacroScore,
6636: )
6637: from farfan_pipeline.utils.paths import safe_join
6638: from farfan_pipeline.core.dependency_lockdown import get_dependency_lockdown
6639: from farfan_pipeline.core.orchestrator import executors_contract as executors
6640: from farfan_pipeline.core.orchestrator.arg_router import (
6641:     ArgRouterError,
6642:     ArgumentValidationError,
6643:     ExtendedArgRouter,
6644: )
6645: from farfan_pipeline.core.orchestrator.class_registry import ClassRegistryError
6646: from farfan_pipeline.core.orchestrator.executor_config import ExecutorConfig
6647: from farfan_pipeline.core.orchestrator.irrigation_synchronizer import (
6648:     IrrigationSynchronizer,
6649:     ExecutionPlan,
6650: )
6651:
6652: logger = logging.getLogger(__name__)
6653: _CORE_MODULE_DIR = Path(__file__).resolve().parent
6654:
6655: # ConfiguraciÃ³n de ambiente
6656: EXPECTED_QUESTION_COUNT = int(os.getenv("EXPECTED_QUESTION_COUNT", "305"))
6657: EXPECTED_METHOD_COUNT = int(os.getenv("EXPECTED_METHOD_COUNT", "416"))
6658: PHASE_TIMEOUT_DEFAULT = int(os.getenv("PHASE_TIMEOUT_SECONDS", "300"))
6659: P01_EXPECTED_CHUNK_COUNT = 60
6660: TIMEOUT_SYNC_PHASES: set[int] = {1}
6661:
6662: P = ParamSpec("P")
6663: T = TypeVar("T")
6664:
```

```
6665:
6666: # ==============================================================================
6667: # UTILIDADES DE PATH
6668: # ==============================================================================
6669:
6670:
6671: def resolve_workspace_path(
6672:     path: str | Path,
6673:     *,
6674:     project_root: Path = PROJECT_ROOT,
6675:     rules_dir: Path = RULES_DIR,
6676:     module_dir: Path = _CORE_MODULE_DIR,
6677: ) -> Path:
6678:     """Resolve repository-relative paths deterministically."""
6679:     path_obj = Path(path)
6680:
6681:     if path_obj.is_absolute():
6682:         return path_obj
6683:
6684:     sanitized = safe_join(project_root, *path_obj.parts)
6685:     candidates = [
6686:         sanitized,
6687:         safe_join(module_dir, *path_obj.parts),
6688:         safe_join(rules_dir, *path_obj.parts),
6689:     ]
6690:
6691:     if not path_obj.parts or path_obj.parts[0] != "rules":
6692:         candidates.append(safe_join(rules_dir, "METODOS", *path_obj.parts))
6693:
6694:     for candidate in candidates:
6695:         if candidate.exists():
6696:             return candidate
6697:
6698:     return sanitized
6699:
6700:
6701: def _normalize_monolith_for_hash(
6702:     monolith: dict[str, Any] | MappingProxyType[str, Any]
6703: ) -> dict[str, Any]:
6704:     """Normalize monolith for hash computation and JSON serialization.
6705:
6706:     Converts MappingProxyType to dict recursively to ensure:
6707:     1. JSON serialization doesn't fail
6708:     2. Hash computation is consistent
6709:     """
6710:     if isinstance(monolith, MappingProxyType):
6711:         monolith = dict(monolith)
6712:
6713:     def _convert(obj: Any) -> Any:
6714:         if isinstance(obj, MappingProxyType):
6715:             obj = dict(obj)
6716:         if isinstance(obj, dict):
6717:             return {k: _convert(v) for k, v in obj.items()}
6718:         if isinstance(obj, list):
6719:             return [_convert(v) for v in obj]
6720:         return obj
```

```
6721:
6722:     normalized = _convert(monolith)
6723:
6724:     try:
6725:         json.dumps(
6726:             normalized, sort_keys=True, ensure_ascii=False, separators=(",", ":")
6727:         )
6728:     except (TypeError, ValueError) as exc:
6729:         raise RuntimeError(f"Monolith normalization failed: {exc}") from exc
6730:
6731:     return normalized
6732:
6733:
6734: # ==========================================================================
6735: # TYPED DICTS Y DATACLASSES
6736: # ==========================================================================
6737:
6738:
6739: class MacroScoreDict(TypedDict):
6740:     """Typed container for macro score evaluation results."""
6741:
6742:     macro_score: MacroScore
6743:     macro_score_normalized: float
6744:     cluster_scores: list[ClusterScore]
6745:     cross_cutting_coherence: float
6746:     systemic_gaps: list[str]
6747:     strategic_alignment: float
6748:     quality_band: str
6749:
6750:
6751: @dataclass
6752: class ClusterScoreData:
6753:     """Type-safe cluster score data for macro evaluation."""
6754:
6755:     id: str
6756:     score: float
6757:     normalized_score: float
6758:
6759:
6760: @dataclass
6761: class MacroEvaluation:
6762:     """Type-safe macro evaluation result."""
6763:
6764:     macro_score: float
6765:     macro_score_normalized: float
6766:     clusters: list[ClusterScoreData]
6767:
6768:
6769: @dataclass
6770: class Evidence:
6771:     """Evidence container for orchestrator results."""
6772:
6773:     modality: str
6774:     elements: list[Any] = field(default_factory=list)
6775:     raw_results: dict[str, Any] = field(default_factory=dict)
6776:
```

```
6777:     def __post_init__(self) -> None:
6778:         if not isinstance(self.modality, str):
6779:             raise TypeError(f"modality must be str, got {type(self.modality)}")
6780:         if not isinstance(self.elements, list):
6781:             raise TypeError(f"elements must be list, got {type(self.elements)}")
6782:         if not isinstance(self.raw_results, dict):
6783:             raise TypeError(f"raw_results must be dict, got {type(self.raw_results)}")
6784:
6785:
6786: @dataclass
6787: class PhaseResult:
6788:     """Result of a single orchestration phase."""
6789:
6790:     success: bool
6791:     phase_id: str
6792:     data: Any
6793:     error: Exception | None
6794:     duration_ms: float
6795:     mode: str
6796:     aborted: bool = False
6797:
6798:     def __post_init__(self) -> None:
6799:         if not isinstance(self.success, bool):
6800:             raise TypeError(f"success must be bool, got {type(self.success)}")
6801:         if not isinstance(self.phase_id, str):
6802:             raise TypeError(f"phase_id must be str, got {type(self.phase_id)}")
6803:         if self.error is not None and not isinstance(self.error, Exception):
6804:             raise TypeError(f"error must be Exception or None, got {type(self.error)}")
6805:         if not isinstance(self.duration_ms, (int, float)):
6806:             raise TypeError(f"duration_ms must be float, got {type(self.duration_ms)}")
6807:         if not isinstance(self.mode, str):
6808:             raise TypeError(f"mode must be str, got {type(self.mode)}")
6809:         if not isinstance(self.aborted, bool):
6810:             raise TypeError(f"aborted must be bool, got {type(self.aborted)}")
6811:
6812:
6813: @dataclass
6814: class MicroQuestionRun:
6815:     """Result of executing a single micro-question."""
6816:
6817:     question_id: str
6818:     question_global: int
6819:     base_slot: str
6820:     metadata: dict[str, Any]
6821:     evidence: Evidence | None
6822:     error: str | None = None
6823:     duration_ms: float | None = None
6824:     aborted: bool = False
6825:
6826:     def __post_init__(self) -> None:
6827:         if not isinstance(self.question_id, str):
6828:             raise TypeError(f"question_id must be str, got {type(self.question_id)}")
6829:         if not isinstance(self.question_global, int):
6830:             raise TypeError(
6831:                 f"question_global must be int, got {type(self.question_global)}"
6832:             )
```

```
        if not isinstance(self.base_slot, str):
            raise TypeError(f"base_slot must be str, got {type(self.base_slot)}")
        if not isinstance(self.metadata, dict):
            raise TypeError(f"metadata must be dict, got {type(self.metadata)}")
        if self.evidence is not None and not isinstance(self.evidence, Evidence):
            raise TypeError(
                f"evidence must be Evidence or None, got {type(self.evidence)}"
            )
        if self.error is not None and not isinstance(self.error, str):
            raise TypeError(f"error must be str or None, got {type(self.error)}")
        if self.duration_ms is not None and not isinstance(
            self.duration_ms, (int, float)
        ):
            raise TypeError(
                f"duration_ms must be float or None, got {type(self.duration_ms)}"
            )
        if not isinstance(self.aborted, bool):
            raise TypeError(f"aborted must be bool, got {type(self.aborted)}")


@dataclass
class ScoredMicroQuestion:
    """Scored micro-question result."""

    question_id: str
    question_global: int
    base_slot: str
    score: float | None
    normalized_score: float | None
    quality_level: str | None
    evidence: Evidence | None
    scoring_details: dict[str, Any] = field(default_factory=dict)
    metadata: dict[str, Any] = field(default_factory=dict)
    error: str | None = None

    def __post_init__(self) -> None:
        if not isinstance(self.question_id, str):
            raise TypeError(f"question_id must be str, got {type(self.question_id)}")
        if not isinstance(self.question_global, int):
            raise TypeError(
                f"question_global must be int, got {type(self.question_global)}"
            )
        if not isinstance(self.base_slot, str):
            raise TypeError(f"base_slot must be str, got {type(self.base_slot)}")
        if self.score is not None and not isinstance(self.score, (int, float)):
            raise TypeError(f"score must be float or None, got {type(self.score)}")
        if self.normalized_score is not None and not isinstance(
            self.normalized_score, (int, float)
        ):
            raise TypeError(
                f"normalized_score must be float or None, got {type(self.normalized_score)}"
            )
        if self.quality_level is not None and not isinstance(self.quality_level, str):
            raise TypeError(
                f"quality_level must be str or None, got {type(self.quality_level)}"
            )
```

```
6889:         if self.evidence is not None and not isinstance(self.evidence, Evidence):
6890:             raise TypeError(
6891:                 f"evidence must be Evidence or None, got {type(self.evidence)}"
6892:             )
6893:         if not isinstance(self.scoring_details, dict):
6894:             raise TypeError(
6895:                 f"scoring_details must be dict, got {type(self.scoring_details)}"
6896:             )
6897:         if not isinstance(self.metadata, dict):
6898:             raise TypeError(f"metadata must be dict, got {type(self.metadata)}")
6899:         if self.error is not None and not isinstance(self.error, str):
6900:             raise TypeError(f"error must be str or None, got {type(self.error)}")
6901:
6902:
6903: # ============================================================================
6904: # MECANISMO DE ABORT
6905: # ============================================================================
6906:
6907:
6908: class AbortRequested(RuntimeError):
6909:     """Raised when an abort signal is triggered during orchestration."""
6910:
6911:     pass
6912:
6913:
6914: class AbortSignal:
6915:     """Thread-safe abort signal shared across orchestration phases."""
6916:
6917:     def __init__(self) -> None:
6918:         self._event = threading.Event()
6919:         self._lock = threading.Lock()
6920:         self._reason: str | None = None
6921:         self._timestamp: datetime | None = None
6922:
6923:     def abort(self, reason: str) -> None:
6924:         """Trigger an abort with a reason and timestamp."""
6925:         if not reason:
6926:             reason = "Abort requested"
6927:         with self._lock:
6928:             if not self._event.is_set():
6929:                 self._event.set()
6930:                 self._reason = reason
6931:                 self._timestamp = datetime.utcnow()
6932:
6933:     def is_aborted(self) -> bool:
6934:         """Check whether abort has been triggered."""
6935:         return self._event.is_set()
6936:
6937:     def get_reason(self) -> str | None:
6938:         """Return the abort reason if set."""
6939:         with self._lock:
6940:             return self._reason
6941:
6942:     def get_timestamp(self) -> datetime | None:
6943:         """Return the abort timestamp if set."""
6944:         with self._lock:
```

```
6945:             return self._timestamp
6946:
6947:     def reset(self) -> None:
6948:         """Clear the abort signal."""
6949:         self._lock.acquire()
6950:         try:
6951:             self._event.clear()
6952:             self._reason = None
6953:             self._timestamp = None
6954:         finally:
6955:             self._lock.release()
6956:
6957:
6958: # ==========================================================================
6959: # GESTIÃ\223N DE RECURSOS
6960: # ==========================================================================
6961:
6962:
6963: class ResourceLimits:
6964:     """Runtime resource guard with adaptive worker prediction."""
6965:
6966:     def __init__(
6967:         self,
6968:         max_memory_mb: float | None = 4096.0,
6969:         max_cpu_percent: float = 85.0,
6970:         max_workers: int = 32,
6971:         min_workers: int = 4,
6972:         hard_max_workers: int = 64,
6973:         history: int = 120,
6974:     ) -> None:
6975:         self.max_memory_mb = max_memory_mb
6976:         self.max_cpu_percent = max_cpu_percent
6977:         self.min_workers = max(1, min_workers)
6978:         self.hard_max_workers = max(self.min_workers, hard_max_workers)
6979:         self._max_workers = max(
6980:             self.min_workers, min(max_workers, self.hard_max_workers)
6981:         )
6982:         self._usage_history: deque[dict[str, float]] = deque(maxlen=history)
6983:         self._semaphore: asyncio.Semaphore | None = None
6984:         self._semaphore_limit = self._max_workers
6985:         self._async_lock: asyncio.Lock | None = None
6986:         self._psutil = None
6987:         self._psutil_process = None
6988:
6989:         try:
6990:             import psutil
6991:
6992:             self._psutil = psutil
6993:             self._psutil_process = psutil.Process(os.getpid())
6994:         except Exception:
6995:             logger.warning("psutil no disponible, usando fallbacks")
6996:
6997:     @property
6998:     def max_workers(self) -> int:
6999:         """Return the current worker budget."""
7000:         return self._max_workers
```

```
7001:
7002:     def attach_semaphore(self, semaphore: asyncio.Semaphore) -> None:
7003:         """Attach an asyncio semaphore for budget control."""
7004:         self._semaphore = semaphore
7005:         self._semaphore_limit = self._max_workers
7006:
7007:     async def apply_worker_budget(self) -> int:
7008:         """Apply the current worker budget to the semaphore."""
7009:         if self._semaphore is None:
7010:             return self._max_workers
7011:
7012:         if self._async_lock is None:
7013:             self._async_lock = asyncio.Lock()
7014:
7015:         async with self._async_lock:
7016:             desired = self._max_workers
7017:             current = self._semaphore_limit
7018:
7019:             if desired > current:
7020:                 for _ in range(desired - current):
7021:                     self._semaphore.release()
7022:             elif desired < current:
7023:                 reduction = current - desired
7024:                 for _ in range(reduction):
7025:                     await self._semaphore.acquire()
7026:
7027:             self._semaphore_limit = desired
7028:             return self._max_workers
7029:
7030:     def _record_usage(self, usage: dict[str, float]) -> None:
7031:         """Record resource usage and predict worker budget."""
7032:         self._usage_history.append(usage)
7033:         self._predict_worker_budget()
7034:
7035:     def _predict_worker_budget(self) -> None:
7036:         """Adjust worker budget based on recent resource usage."""
7037:         if len(self._usage_history) < 5:
7038:             return
7039:
7040:         recent_cpu = [e["cpu_percent"] for e in list(self._usage_history)[-5:]]
7041:         recent_mem = [e["memory_percent"] for e in list(self._usage_history)[-5:]]
7042:
7043:         avg_cpu = statistics.mean(recent_cpu)
7044:         avg_mem = statistics.mean(recent_mem)
7045:
7046:         new_budget = self._max_workers
7047:
7048:         if (self.max_cpu_percent and avg_cpu > self.max_cpu_percent * 0.95) or (
7049:             self.max_memory_mb and avg_mem > 90.0
7050:         ):
7051:             new_budget = max(self.min_workers, self._max_workers - 1)
7052:         elif avg_cpu < self.max_cpu_percent * 0.6 and avg_mem < 70.0:
7053:             new_budget = min(self.hard_max_workers, self._max_workers + 1)
7054:
7055:         self._max_workers = max(
7056:             self.min_workers, min(new_budget, self.hard_max_workers)
```

```
        )

    def get_resource_usage(self) -> dict[str, Any]:
        """Capture current resource usage metrics."""
        timestamp = datetime.utcnow().isoformat()
        cpu_percent = 0.0
        memory_percent = 0.0
        rss_mb = 0.0

        if self._psutil:
            try:
                cpu_percent = float(self._psutil.cpu_percent(interval=None))
                virtual_memory = self._psutil.virtual_memory()
                memory_percent = float(virtual_memory.percent)
                if self._psutil_process:
                    rss_mb = float(
                        self._psutil_process.memory_info().rss / (1024 * 1024)
                    )
            except Exception:
                cpu_percent = 0.0
        else:
            try:
                load1, _, _ = os.getloadavg()
                cpu_percent = float(min(100.0, load1 * 100))
            except OSError:
                cpu_percent = 0.0

            try:
                import resource

                usage_info = resource.getrusage(resource.RUSAGE_SELF)
                rss_mb = float(usage_info.ru_maxrss / 1024)
            except Exception:
                rss_mb = 0.0

        usage_dict: dict[str, float] = {
            "cpu_percent": cpu_percent,
            "memory_percent": memory_percent,
            "rss_mb": rss_mb,
            "worker_budget": float(self._max_workers),
        }
        usage: dict[str, Any] = {"timestamp": timestamp, **usage_dict}

        self._record_usage(usage)
        return usage

    def check_memory_exceeded(
        self, usage: dict[str, Any] | None = None
    ) -> tuple[bool, dict[str, Any]]:
        """Check if memory limit has been exceeded."""
        usage = usage or self.get_resource_usage()
        exceeded = False
        if self.max_memory_mb is not None:
            exceeded = usage.get("rss_mb", 0.0) > self.max_memory_mb
        return exceeded, usage

```

```
    def check_cpu_exceeded(
        self, usage: dict[str, Any] | None = None
    ) -> tuple[bool, dict[str, Any]]:
        """Check if CPU limit has been exceeded."""
        usage = usage or self.get_resource_usage()
        exceeded = False
        if self.max_cpu_percent:
            exceeded = usage.get("cpu_percent", 0.0) > self.max_cpu_percent
        return exceeded, usage

    def get_usage_history(self) -> list[dict[str, float]]:
        """Return the recorded usage history."""
        return list(self._usage_history)


# ============================================================================
# INSTRUMENTACIÃ\223N DE FASES
# ============================================================================


class PhaseInstrumentation:
    """Collects granular telemetry for each orchestration phase."""

    def __init__(
        self,
        phase_id: int,
        name: str,
        items_total: int | None = None,
        snapshot_interval: int = 10,
        resource_limits: ResourceLimits | None = None,
    ) -> None:
        self.phase_id = phase_id
        self.name = name
        self.items_total = items_total or 0
        self.snapshot_interval = max(1, snapshot_interval)
        self.resource_limits = resource_limits
        self.items_processed = 0
        self.start_time: float | None = None
        self.end_time: float | None = None
        self.warnings: list[dict[str, Any]] = []
        self.errors: list[dict[str, Any]] = []
        self.resource_snapshots: list[dict[str, Any]] = []
        self.latencies: list[float] = []
        self.anomalies: list[dict[str, Any]] = []

    def start(self, items_total: int | None = None) -> None:
        """Mark the start of phase execution."""
        if items_total is not None:
            self.items_total = items_total
        self.start_time = time.perf_counter()

    def increment(self, count: int = 1, latency: float | None = None) -> None:
        """Increment processed item count and optionally record latency."""
        self.items_processed += count
        if latency is not None:
            self.latencies.append(latency)
```

```
            self._detect_latency_anomaly(latency)
        if self.resource_limits and self.should_snapshot():
            self.capture_resource_snapshot()

    def should_snapshot(self) -> bool:
        """Determine if a resource snapshot should be captured."""
        if self.items_total == 0 or self.items_processed == 0:
            return False
        return self.items_processed % self.snapshot_interval == 0

    def capture_resource_snapshot(self) -> None:
        """Capture a resource usage snapshot."""
        if not self.resource_limits:
            return
        snapshot = self.resource_limits.get_resource_usage()
        snapshot["items_processed"] = self.items_processed
        self.resource_snapshots.append(snapshot)

    def record_warning(self, category: str, message: str, **extra: Any) -> None:
        """Record a warning during phase execution."""
        entry = {
            "category": category,
            "message": message,
            **extra,
            "timestamp": datetime.utcnow().isoformat(),
        }
        self.warnings.append(entry)

    def record_error(self, category: str, message: str, **extra: Any) -> None:
        """Record an error during phase execution."""
        entry = {
            "category": category,
            "message": message,
            **extra,
            "timestamp": datetime.utcnow().isoformat(),
        }
        self.errors.append(entry)

    def _detect_latency_anomaly(self, latency: float) -> None:
        """Detect latency anomalies using statistical thresholds."""
        if len(self.latencies) < 5:
            return

        mean_latency = statistics.mean(self.latencies)
        std_latency = statistics.pstdev(self.latencies) or 0.0
        threshold = mean_latency + (3 * std_latency)

        if std_latency and latency > threshold:
            self.anomalies.append(
                {
                    "type": "latency_spike",
                    "latency": latency,
                    "mean": mean_latency,
                    "std": std_latency,
                    "timestamp": datetime.utcnow().isoformat(),
                }
```

```
            )

    def complete(self) -> None:
        """Mark the end of phase execution."""
        self.end_time = time.perf_counter()

    def duration_ms(self) -> float | None:
        """Return the phase duration in milliseconds."""
        if self.start_time is None or self.end_time is None:
            return None
        return (self.end_time - self.start_time) * 1000.0

    def progress(self) -> float | None:
        """Return the progress fraction (0.0 to 1.0)."""
        if not self.items_total:
            return None
        return min(1.0, self.items_processed / float(self.items_total))

    def throughput(self) -> float | None:
        """Return items processed per second."""
        if self.start_time is None:
            return None
        elapsed = (
            (time.perf_counter() - self.start_time)
            if self.end_time is None
            else (self.end_time - self.start_time)
        )
        if not elapsed:
            return None
        return self.items_processed / elapsed

    def latency_histogram(self) -> dict[str, float | None]:
        """Return latency percentiles."""
        if not self.latencies:
            return {"p50": None, "p95": None, "p99": None}

        sorted_latencies = sorted(self.latencies)

        def percentile(p: float) -> float:
            if not sorted_latencies:
                return 0.0
            k = (len(sorted_latencies) - 1) * (p / 100.0)
            f = int(k)
            c = min(f + 1, len(sorted_latencies) - 1)
            if f == c:
                return sorted_latencies[int(k)]
            d0 = sorted_latencies[f] * (c - k)
            d1 = sorted_latencies[c] * (k - f)
            return d0 + d1

        return {
            "p50": percentile(50.0),
            "p95": percentile(95.0),
            "p99": percentile(99.0),
        }

```

```
7281:     def build_metrics(self) -> dict[str, Any]:
7282:         """Build a metrics summary dictionary."""
7283:         return {
7284:             "phase_id": self.phase_id,
7285:             "name": self.name,
7286:             "duration_ms": self.duration_ms(),
7287:             "items_processed": self.items_processed,
7288:             "items_total": self.items_total,
7289:             "progress": self.progress(),
7290:             "throughput": self.throughput(),
7291:             "warnings": list(self.warnings),
7292:             "errors": list(self.errors),
7293:             "resource_snapshots": list(self.resource_snapshots),
7294:             "latency_histogram": self.latency_histogram(),
7295:             "anomalies": list(self.anomalies),
7296:         }
7297:
7298:
7299: # ============================================================================
7300: # TIMEOUT DE FASES
7301: # ============================================================================
7302:
7303:
7304: class PhaseTimeoutError(RuntimeError):
7305:     """Raised when a phase exceeds its timeout."""
7306:
7307:     def __init__(self, phase_id: int | str, phase_name: str, timeout_s: float) -> None:
7308:         self.phase_id = phase_id
7309:         self.phase_name = phase_name
7310:         self.timeout_s = timeout_s
7311:         super().__init__(
7312:             f"Phase {phase_id} ({phase_name}) timed out after {timeout_s}s"
7313:         )
7314:
7315:
7316: async def execute_phase_with_timeout(
7317:     phase_id: int,
7318:     phase_name: str,
7319:     timeout_s: float = 300.0,
7320:     coro: Callable[..., T] | None = None,
7321:     handler: Callable[..., T] | None = None,
7322:     args: tuple[Any, ...] | None = None,
7323:     kwargs: dict[str, Any] | None = None,
7324: ) -> T:
7325:     """Execute an async phase with timeout and comprehensive logging."""
7326:     target = coro or handler
7327:     if target is None:
7328:         raise ValueError("Either 'coro' or 'handler' must be provided")
7329:
7330:     call_args = args or ()
7331:     call_kwargs = kwargs or {}
7332:
7333:     start = time.perf_counter()
7334:     logger.info(f"Fase {phase_id} ({phase_name}) iniciada, timeout={timeout_s}s")
7335:
7336:     try:
```

```
7337:         result = await asyncio.wait_for(
7338:             target(*call_args, **call_kwargs), timeout=timeout_s
7339:         )
7340:         elapsed = time.perf_counter() - start
7341:         logger.info(f"Fase {phase_id} completada en {elapsed:.2f}s")
7342:         return result
7343:
7344:     except asyncio.TimeoutError as exc:
7345:         elapsed = time.perf_counter() - start
7346:         logger.error(f"Fase {phase_id} TIMEOUT despuÃ©s de {elapsed:.2f}s")
7347:         raise PhaseTimeoutError(phase_id, phase_name, timeout_s) from exc
7348:
7349:     except asyncio.CancelledError:
7350:         elapsed = time.perf_counter() - start
7351:         logger.warning(f"Fase {phase_id} CANCELADA despuÃ©s de {elapsed:.2f}s")
7352:         raise
7353:
7354:     except Exception as exc:
7355:         elapsed = time.perf_counter() - start
7356:         logger.error(
7357:             f"Fase {phase_id} ERROR: {exc} (despuÃ©s de {elapsed:.2f}s)", exc_info=True
7358:         )
7359:         raise
7360:
7361:
7362: # ============================================================================
7363: # METHOD EXECUTOR Y LAZY LOADING
7364: # ============================================================================
7365:
7366:
7367: class _LazyInstanceDict:
7368:     """Lazy instance dictionary for backward compatibility."""
7369:
7370:     def __init__(self, method_registry: Any) -> None:
7371:         self._registry = method_registry
7372:
7373:     def get(self, class_name: str, default: Any = None) -> Any:
7374:         """Get instance lazily."""
7375:         try:
7376:             return self._registry._get_instance(class_name)
7377:         except Exception:
7378:             return default
7379:
7380:     def __getitem__(self, class_name: str) -> Any:
7381:         """Get instance lazily (dict access)."""
7382:         return self._registry._get_instance(class_name)
7383:
7384:     def __contains__(self, class_name: str) -> bool:
7385:         """Check if class is available."""
7386:         return class_name in self._registry._class_paths
7387:
7388:     def keys(self) -> list[str]:
7389:         """Get available class names."""
7390:         return list(self._registry._class_paths.keys())
7391:
7392:     def values(self) -> list[Any]:
```

```
7393:         """Get instantiated instances (triggers lazy loading)."""
7394:         return [self.get(name) for name in self.keys()]
7395:
7396:     def items(self) -> list[tuple[str, Any]]:
7397:         """Get (name, instance) pairs (triggers lazy loading)."""
7398:         return [(name, self.get(name)) for name in self.keys()]
7399:
7400:     def __len__(self) -> int:
7401:         """Get number of available classes."""
7402:         return len(self._registry._class_paths)
7403:
7404:
7405: class MethodExecutor:
7406:     """Execute catalog methods using lazy method injection."""
7407:
7408:     def __init__(
7409:         self,
7410:         dispatcher: Any | None = None,
7411:         signal_registry: Any | None = None,
7412:         method_registry: Any | None = None,
7413:     ) -> None:
7414:         from farfan_pipeline.core.orchestrator.method_registry import (
7415:             MethodRegistry,
7416:             setup_default_instantiation_rules,
7417:         )
7418:
7419:         self.degraded_mode = False
7420:         self.degraded_reasons: list[str] = []
7421:         self.signal_registry = signal_registry
7422:
7423:         if method_registry is not None:
7424:             self._method_registry = method_registry
7425:         else:
7426:             try:
7427:                 self._method_registry = MethodRegistry()
7428:                 setup_default_instantiation_rules(self._method_registry)
7429:                 logger.info("method_registry_initialized_lazy_mode")
7430:             except Exception as exc:
7431:                 self.degraded_mode = True
7432:                 reason = f"Method registry initialization failed: {exc}"
7433:                 self.degraded_reasons.append(reason)
7434:                 logger.error("DEGRADED MODE: %s", reason)
7435:                 self._method_registry = MethodRegistry(class_paths={})
7436:
7437:         try:
7438:             from farfan_pipeline.core.orchestrator.class_registry import (
7439:                 build_class_registry,
7440:             )
7441:
7442:             registry = build_class_registry()
7443:         except (ClassRegistryError, ModuleNotFoundError, ImportError) as exc:
7444:             self.degraded_mode = True
7445:             reason = f"Could not build class registry: {exc}"
7446:             self.degraded_reasons.append(reason)
7447:             logger.warning("DEGRADED MODE: %s", reason)
7448:             registry = {}
```

```
7449:
7450:         self._router = ExtendedArgRouter(registry)
7451:         self.instances = _LazyInstanceDict(self._method_registry)
7452:
7453:     @staticmethod
7454:     def _supports_parameter(callable_obj: Any, parameter_name: str) -> bool:
7455:         try:
7456:             signature = inspect.signature(callable_obj)
7457:         except (TypeError, ValueError):
7458:             return False
7459:         return parameter_name in signature.parameters
7460:
7461:     def execute(self, class_name: str, method_name: str, **kwargs: Any) -> Any:
7462:         """Execute a method using lazy instantiation."""
7463:         from farfan_pipeline.core.orchestrator.method_registry import (
7464:             MethodRegistryError,
7465:         )
7466:
7467:         try:
7468:             method = self._method_registry.get_method(class_name, method_name)
7469:         except MethodRegistryError as exc:
7470:             logger.error(f"method_retrieval_failed: {class_name}.{method_name}: {exc}")
7471:             if self.degraded_mode:
7472:                 logger.warning("Returning None due to degraded mode")
7473:                 return None
7474:             raise AttributeError(
7475:                 f"Cannot retrieve {class_name}.{method_name}: {exc}"
7476:             ) from exc
7477:
7478:         try:
7479:             args, routed_kwargs = self._router.route(
7480:                 class_name, method_name, dict(kwargs)
7481:             )
7482:             return method(*args, **routed_kwargs)
7483:         except (ArgRouterError, ArgumentValidationError):
7484:             logger.exception(f"Argument routing failed for {class_name}.{method_name}")
7485:             raise
7486:         except Exception:
7487:             logger.exception(f"Method execution failed for {class_name}.{method_name}")
7488:             raise
7489:
7490:     def inject_method(
7491:         self, class_name: str, method_name: str, method: Callable[..., Any]
7492:     ) -> None:
7493:         """Inject a method directly without requiring a class."""
7494:         self._method_registry.inject_method(class_name, method_name, method)
7495:         logger.info(f"method_injected_into_executor: {class_name}.{method_name}")
7496:
7497:     def has_method(self, class_name: str, method_name: str) -> bool:
7498:         """Check if a method is available."""
7499:         return self._method_registry.has_method(class_name, method_name)
7500:
7501:     def get_registry_stats(self) -> dict[str, Any]:
7502:         """Get statistics from the method registry."""
7503:         return self._method_registry.get_stats()
7504:
```

```
7505:     def get_routing_metrics(self) -> dict[str, Any]:
7506:         """Get routing metrics from ExtendedArgRouter."""
7507:         if hasattr(self._router, "get_metrics"):
7508:             return self._router.get_metrics()
7509:         return {}
7510:
7511:
7512: # ==========================================================================
7513: # VALIDACIÃ\223N DE DEFINICIONES DE FASES
7514: # ==========================================================================
7515:
7516:
7517: def validate_phase_definitions(
7518:     phase_list: list[tuple[int, str, str, str]], orchestrator_class: type
7519: ) -> None:
7520:     """Validate phase definitions for structural coherence."""
7521:     if not phase_list:
7522:         raise RuntimeError(
7523:             "FASES cannot be empty - no phases defined for orchestration"
7524:         )
7525:
7526:     phase_ids = [phase[0] for phase in phase_list]
7527:
7528:     seen_ids = set()
7529:     for phase_id in phase_ids:
7530:         if phase_id in seen_ids:
7531:             raise RuntimeError(f"Duplicate phase ID {phase_id} in FASES definition")
7532:         seen_ids.add(phase_id)
7533:
7534:     if phase_ids != sorted(phase_ids):
7535:         raise RuntimeError(
7536:             f"Phase IDs must be sorted in ascending order. Got {phase_ids}"
7537:         )
7538:     if phase_ids[0] != 0:
7539:         raise RuntimeError(f"Phase IDs must start from 0. Got first ID: {phase_ids[0]}")
7540:     if phase_ids[-1] != len(phase_list) - 1:
7541:         raise RuntimeError(
7542:             f"Phase IDs must be contiguous from 0 to {len(phase_list) - 1}. Got highest ID: {phase_ids[-1]}"
7543:         )
7544:
7545:     valid_modes = {"sync", "async"}
7546:     for phase_id, mode, handler_name, label in phase_list:
7547:         if mode not in valid_modes:
7548:             raise RuntimeError(
7549:                 f"Phase {phase_id} ({label}): invalid mode '{mode}'. Mode must be one of {valid_modes}"
7550:             )
7551:
7552:         if not hasattr(orchestrator_class, handler_name):
7553:             raise RuntimeError(
7554:                 f"Phase {phase_id} ({label}): handler method '{handler_name}' does not exist in {orchestrator_class.__name__}"
7555:             )
7556:
7557:         handler = getattr(orchestrator_class, handler_name, None)
7558:         if not callable(handler):
7559:             raise RuntimeError(
7560:                 f"Phase {phase_id} ({label}): handler '{handler_name}' is not callable"
```

```
7561:             )
7562:
7563:
7564: # ============================================================================
7565: # ORQUESTADOR PRINCIPAL
7566: # ============================================================================
7567:
7568:
7569: class Orchestrator:
7570:     """Robust 11-phase orchestrator with abort support and resource control."""
7571:
7572:     FASES: list[tuple[int, str, str, str]] = [
7573:         (0, "sync", "_load_configuration", "FASE 0 - ValidaciÃ³n de ConfiguraciÃ³n"),
7574:         (1, "sync", "_ingest_document", "FASE 1 - IngestiÃ³n de Documento"),
7575:         (2, "async", "_execute_micro_questions_async", "FASE 2 - Micro Preguntas"),
7576:         (3, "async", "_score_micro_results_async", "FASE 3 - Scoring Micro"),
7577:         (4, "async", "_aggregate_dimensions_async", "FASE 4 - AgregaciÃ³n Dimensiones"),
7578:         (5, "async", "_aggregate_policy_areas_async", "FASE 5 - AgregaciÃ³n Ã\201reas"),
7579:         (6, "sync", "_aggregate_clusters", "FASE 6 - AgregaciÃ³n ClÃºsteres"),
7580:         (7, "sync", "_evaluate_macro", "FASE 7 - EvaluaciÃ³n Macro"),
7581:         (8, "async", "_generate_recommendations", "FASE 8 - Recomendaciones"),
7582:         (9, "sync", "_assemble_report", "FASE 9 - Ensamblado de Reporte"),
7583:         (10, "async", "_format_and_export", "FASE 10 - Formateo y ExportaciÃ³n"),
7584:     ]
7585:
7586:     PHASE_ITEM_TARGETS: dict[int, int] = {
7587:         0: 1,
7588:         1: 1,
7589:         2: 300,
7590:         3: 300,
7591:         4: 60,
7592:         5: 10,
7593:         6: 4,
7594:         7: 1,
7595:         8: 1,
7596:         9: 1,
7597:         10: 1,
7598:     }
7599:
7600:     PHASE_OUTPUT_KEYS: dict[int, str] = {
7601:         0: "config",
7602:         1: "document",
7603:         2: "micro_results",
7604:         3: "scored_results",
7605:         4: "dimension_scores",
7606:         5: "policy_area_scores",
7607:         6: "cluster_scores",
7608:         7: "macro_result",
7609:         8: "recommendations",
7610:         9: "report",
7611:         10: "export_payload",
7612:     }
7613:
7614:     PHASE_ARGUMENT_KEYS: dict[int, list[str]] = {
7615:         1: ["pdf_path", "config"],
7616:         2: ["document", "config"],
```

```
7617:         3: ["micro_results", "config"],
7618:         4: ["scored_results", "config"],
7619:         5: ["dimension_scores", "config"],
7620:         6: ["policy_area_scores", "config"],
7621:         7: ["cluster_scores", "config"],
7622:         8: ["macro_result", "config"],
7623:         9: ["recommendations", "config"],
7624:         10: ["report", "config"],
7625:     }
7626:
7627:     PHASE_TIMEOUTS: dict[int, float] = {
7628:         0: 60,
7629:         1: 120,
7630:         2: 600,
7631:         3: 300,
7632:         4: 180,
7633:         5: 120,
7634:         6: 60,
7635:         7: 60,
7636:         8: 120,
7637:         9: 60,
7638:         10: 120,
7639:     }
7640:
7641:     PERCENTAGE_SCALE: int = 100
7642:
7643:     def __init__(
7644:         self,
7645:         method_executor: MethodExecutor,
7646:         questionnaire: "CanonicalQuestionnaire",
7647:         executor_config: ExecutorConfig,
7648:         calibration_orchestrator: Any | None = None,
7649:         resource_limits: ResourceLimits | None = None,
7650:         resource_snapshot_interval: int = 10,
7651:         recommendation_engine_port: RecommendationEnginePort | None = None,
7652:         processor_bundle: Any | None = None,
7653:     ) -> None:
7654:         """Initialize the orchestrator with all dependencies injected.
7655:
7656:         Args:
7657:             method_executor: Configured MethodExecutor instance
7658:             questionnaire: Loaded and validated CanonicalQuestionnaire
7659:             executor_config: Executor configuration object
7660:             calibration_orchestrator: Calibration orchestrator instance
7661:             resource_limits: Resource limit configuration
7662:             resource_snapshot_interval: Interval for resource snapshots
7663:             recommendation_engine_port: Optional recommendation engine port
7664:             processor_bundle: ProcessorBundle with enriched_signal_packs (WIRING REQUIRED)
7665:         """
7666:         from farfan_pipeline.core.orchestrator.questionnaire import (
7667:             _validate_questionnaire_schema,
7668:             get_questionnaire_provider,
7669:         )
7670:
7671:         validate_phase_definitions(self.FASES, self.__class__)
7672:
```

```
7673:         self.executor = method_executor
7674:         self._canonical_questionnaire = questionnaire
7675:         self._monolith_data = dict(questionnaire.data)
7676:         self.executor_config = executor_config
7677:         self.calibration_orchestrator = calibration_orchestrator
7678:         self.resource_limits = resource_limits or ResourceLimits()
7679:         self.resource_snapshot_interval = max(1, resource_snapshot_interval)
7680:         self.questionnaire_provider = get_questionnaire_provider()
7681:
7682:         # ========================================================================
7683:         # WIRING: ProcessorBundle â\206\222 Orchestrator
7684:         # Store enriched_signal_packs for use in Phase 2
7685:         # ========================================================================
7686:         self._processor_bundle = processor_bundle
7687:         self._enriched_packs = None
7688:         if processor_bundle is not None:
7689:             if hasattr(processor_bundle, "enriched_signal_packs"):
7690:                 self._enriched_packs = processor_bundle.enriched_signal_packs
7691:                 logger.info(
7692:                     f"Orchestrator wired with {len(self._enriched_packs) if self._enriched_packs else 0} enriched signal packs"
7693:                 )
7694:             else:
7695:                 logger.warning(
7696:                     "ProcessorBundle provided but missing enriched_signal_packs attribute"
7697:                 )
7698:         else:
7699:             logger.warning(
7700:                 "No ProcessorBundle provided - signal enrichment unavailable"
7701:             )
7702:
7703:         # ========================================================================
7704:         # WIRING: MethodExecutor.signal_registry verification
7705:         # Ensure MethodExecutor has access to QuestionnaireSignalRegistry
7706:         # ========================================================================
7707:         if (
7708:             not hasattr(self.executor, "signal_registry")
7709:             or self.executor.signal_registry is None
7710:         ):
7711:             logger.error("WIRING VIOLATION: MethodExecutor missing signal_registry")
7712:             raise RuntimeError(
7713:                 "MethodExecutor must be configured with signal_registry. "
7714:                 "Executors require indirect access to QuestionnaireSignalRegistry."
7715:             )
7716:         else:
7717:             logger.info("â\234\223 MethodExecutor.signal_registry verified")
7718:
7719:         try:
7720:             _validate_questionnaire_schema(self._monolith_data)
7721:         except (ValueError, TypeError) as e:
7722:             raise RuntimeError(f"Questionnaire structure validation failed: {e}") from e
7723:
7724:         if not self.executor.instances:
7725:             raise RuntimeError(
7726:                 "MethodExecutor.instances is empty - no executable methods registered"
7727:             )
7728:
```

```
        # Executor registry
        self.executors = {
            "D1-Q1": executors.D1Q1_Executor,
            "D1-Q2": executors.D1Q2_Executor,
            "D1-Q3": executors.D1Q3_Executor,
            "D1-Q4": executors.D1Q4_Executor,
            "D1-Q5": executors.D1Q5_Executor,
            "D2-Q1": executors.D2Q1_Executor,
            "D2-Q2": executors.D2Q2_Executor,
            "D2-Q3": executors.D2Q3_Executor,
            "D2-Q4": executors.D2Q4_Executor,
            "D2-Q5": executors.D2Q5_Executor,
            "D3-Q1": executors.D3Q1_Executor,
            "D3-Q2": executors.D3Q2_Executor,
            "D3-Q3": executors.D3Q3_Executor,
            "D3-Q4": executors.D3Q4_Executor,
            "D3-Q5": executors.D3Q5_Executor,
            "D4-Q1": executors.D4Q1_Executor,
            "D4-Q2": executors.D4Q2_Executor,
            "D4-Q3": executors.D4Q3_Executor,
            "D4-Q4": executors.D4Q4_Executor,
            "D4-Q5": executors.D4Q5_Executor,
            "D5-Q1": executors.D5Q1_Executor,
            "D5-Q2": executors.D5Q2_Executor,
            "D5-Q3": executors.D5Q3_Executor,
            "D5-Q4": executors.D5Q4_Executor,
            "D5-Q5": executors.D5Q5_Executor,
            "D6-Q1": executors.D6Q1_Executor,
            "D6-Q2": executors.D6Q2_Executor,
            "D6-Q3": executors.D6Q3_Executor,
            "D6-Q4": executors.D6Q4_Executor,
            "D6-Q5": executors.D6Q5_Executor,
        }

        # State management
        self.abort_signal = AbortSignal()
        self.phase_results: list[PhaseResult] = []
        self._phase_instrumentation: dict[int, PhaseInstrumentation] = {}
        self._phase_status: dict[int, str] = {
            phase_id: "not_started" for phase_id, *_ in self.FASES
        }
        self._phase_outputs: dict[int, Any] = {}
        self._context: dict[str, Any] = {}
        self._start_time: float | None = None
        self._execution_plan: ExecutionPlan | None = None

        # Additional attributes for phase 0 validations
        self._method_map_data: dict[str, Any] | None = None
        self._schema_data: dict[str, Any] | None = None
        self.catalog: dict[str, Any] | None = None

        self.dependency_lockdown = get_dependency_lockdown()
        logger.info(
            f"Orchestrator dependency mode: {self.dependency_lockdown.get_mode_description()}"
        )

```

```
7785:         self.recommendation_engine = recommendation_engine_port
7786:         if self.recommendation_engine is not None:
7787:             logger.info("RecommendationEngine port injected successfully")
7788:         else:
7789:             logger.warning(
7790:                 "No RecommendationEngine port provided - recommendations unavailable"
7791:             )
7792:
7793:     def _ensure_not_aborted(self) -> None:
7794:         """Check if orchestration has been aborted and raise exception if so."""
7795:         if self.abort_signal.is_aborted():
7796:             reason = self.abort_signal.get_reason() or "Unknown reason"
7797:             raise AbortRequested(f"Orchestration aborted: {reason}")
7798:
7799:     def request_abort(self, reason: str) -> None:
7800:         """Request orchestration to abort with a specific reason."""
7801:         self.abort_signal.abort(reason)
7802:
7803:     def reset_abort(self) -> None:
7804:         """Reset the abort signal to allow new orchestration runs."""
7805:         self.abort_signal.reset()
7806:
7807:     def _get_phase_timeout(self, phase_id: int) -> float:
7808:         """Get timeout for a specific phase."""
7809:         return self.PHASE_TIMEOUTS.get(phase_id, 300.0)
7810:
7811:     def _resolve_path(self, path: str | None) -> str | None:
7812:         """Resolve a relative or absolute path."""
7813:         if path is None:
7814:             return None
7815:         resolved = resolve_workspace_path(path)
7816:         return str(resolved)
7817:
7818:     async def process_development_plan_async(
7819:         self, pdf_path: str, preprocessed_document: Any | None = None
7820:     ) -> list[PhaseResult]:
7821:         """Execute the complete 11-phase pipeline asynchronously."""
7822:         self.reset_abort()
7823:         self.phase_results = []
7824:         self._phase_instrumentation = {}
7825:         self._phase_outputs = {}
7826:         self._context = {"pdf_path": pdf_path}
7827:
7828:         if preprocessed_document is not None:
7829:             self._context["preprocessed_override"] = preprocessed_document
7830:
7831:         self._phase_status = {phase_id: "not_started" for phase_id, *_ in self.FASES}
7832:         self._start_time = time.perf_counter()
7833:
7834:         for phase_id, mode, handler_name, phase_label in self.FASES:
7835:             self._ensure_not_aborted()
7836:
7837:             handler = getattr(self, handler_name)
7838:             instrumentation = PhaseInstrumentation(
7839:                 phase_id=phase_id,
7840:                 name=phase_label,
```

```
                items_total=self.PHASE_ITEM_TARGETS.get(phase_id),
                snapshot_interval=self.resource_snapshot_interval,
                resource_limits=self.resource_limits,
            )

            instrumentation.start(items_total=self.PHASE_ITEM_TARGETS.get(phase_id))
            self._phase_instrumentation[phase_id] = instrumentation
            self._phase_status[phase_id] = "running"

            args = [
                self._context[key] for key in self.PHASE_ARGUMENT_KEYS.get(phase_id, [])
            ]

            success = False
            data: Any = None
            error: Exception | None = None

            try:
                if mode == "sync":
                    if phase_id in TIMEOUT_SYNC_PHASES:
                        data = await execute_phase_with_timeout(
                            phase_id=phase_id,
                            phase_name=phase_label,
                            timeout_s=self._get_phase_timeout(phase_id),
                            coro=asyncio.to_thread,
                            args=(handler,) + tuple(args),
                        )
                    else:
                        data = handler(*args)
                else:
                    data = await execute_phase_with_timeout(
                        phase_id=phase_id,
                        phase_name=phase_label,
                        timeout_s=self._get_phase_timeout(phase_id),
                        handler=handler,
                        args=tuple(args),
                    )
                success = True

            except PhaseTimeoutError as exc:
                error = exc
                instrumentation.record_error("timeout", str(exc))
                self.request_abort(f"Fase {phase_id} timed out: {exc}")

            except AbortRequested as exc:
                error = exc
                instrumentation.record_warning("abort", str(exc))

            except Exception as exc:
                logger.exception(f"Fase {phase_label} fallÃ³")
                error = exc
                instrumentation.record_error("exception", str(exc))
                self.request_abort(f"Fase {phase_id} fallÃ³: {exc}")

            finally:
                instrumentation.complete()
```

```
7897:
7898:             aborted = self.abort_signal.is_aborted()
7899:             duration_ms = instrumentation.duration_ms() or 0.0
7900:
7901:             phase_result = PhaseResult(
7902:                 success=success and not aborted,
7903:                 phase_id=str(phase_id),
7904:                 data=data,
7905:                 error=error,
7906:                 duration_ms=duration_ms,
7907:                 mode=mode,
7908:                 aborted=aborted,
7909:             )
7910:             self.phase_results.append(phase_result)
7911:
7912:             if success and not aborted:
7913:                 self._phase_outputs[phase_id] = data
7914:                 out_key = self.PHASE_OUTPUT_KEYS.get(phase_id)
7915:                 if out_key:
7916:                     self._context[out_key] = data
7917:                 self._phase_status[phase_id] = "completed"
7918:
7919:                 # Build execution plan after Phase 1
7920:                 if phase_id == 1:
7921:                     try:
7922:                         logger.info("Building execution plan after Phase 1 completion")
7923:                         document = self._context.get("document")
7924:                         chunks = getattr(document, "chunks", []) if document else []
7925:
7926:                         synchronizer = IrrigationSynchronizer(
7927:                             questionnaire=self._monolith_data, document_chunks=chunks
7928:                         )
7929:                         self._execution_plan = synchronizer.build_execution_plan()
7930:
7931:                         logger.info(
7932:                             f"Execution plan built: {len(self._execution_plan.tasks)} tasks, "
7933:                             f"plan_id={self._execution_plan.plan_id}"
7934:                         )
7935:                     except ValueError as e:
7936:                         logger.error(f"Failed to build execution plan: {e}")
7937:                         self.request_abort(f"Synchronization failed: {e}")
7938:                         raise
7939:             elif aborted:
7940:                 self._phase_status[phase_id] = "aborted"
7941:                 break
7942:             else:
7943:                 self._phase_status[phase_id] = "failed"
7944:                 break
7945:
7946:         return self.phase_results
7947:
7948:     def process_development_plan(
7949:         self, pdf_path: str, preprocessed_document: Any | None = None
7950:     ) -> list[PhaseResult]:
7951:         """Sync wrapper for process_development_plan_async."""
7952:         try:
```

```
            loop = asyncio.get_running_loop()
        except RuntimeError:
            loop = None

        if loop and loop.is_running():
            raise RuntimeError(
                "process_development_plan() must be called outside an active asyncio loop"
            )

        return asyncio.run(
            self.process_development_plan_async(
                pdf_path, preprocessed_document=preprocessed_document
            )
        )

    async def process(self, preprocessed_document: Any) -> list[PhaseResult]:
        """DEPRECATED ALIAS for process_development_plan_async()."""
        import warnings

        warnings.warn(
            "Orchestrator.process() is deprecated. Use process_development_plan_async() instead.",
            DeprecationWarning,
            stacklevel=2,
        )

        pdf_path = getattr(preprocessed_document, "source_path", None)
        if pdf_path is None:
            metadata = getattr(preprocessed_document, "metadata", {})
            pdf_path = metadata.get("source_path", "unknown.pdf")

        return await self.process_development_plan_async(
            pdf_path=str(pdf_path), preprocessed_document=preprocessed_document
        )

    def get_processing_status(self) -> dict[str, Any]:
        """Get current processing status."""
        if self._start_time is None:
            status = "not_started"
            elapsed = 0.0
            completed_flag = False
        else:
            aborted = self.abort_signal.is_aborted()
            status = "aborted" if aborted else "running"
            elapsed = time.perf_counter() - self._start_time
            completed_flag = (
                all(state == "completed" for state in self._phase_status.values())
                and not aborted
            )

        completed = sum(
            1 for state in self._phase_status.values() if state == "completed"
        )
        total = len(self.FASES)
        overall_progress = completed / total if total else 0.0

        phase_progress = {
```

```
            str(phase_id): instr.progress()
            for phase_id, instr in self._phase_instrumentation.items()
        }

        resource_usage = (
            self.resource_limits.get_resource_usage() if self._start_time else {}
        )

        return {
            "status": status,
            "overall_progress": overall_progress,
            "phase_progress": phase_progress,
            "elapsed_time_s": elapsed,
            "resource_usage": resource_usage,
            "abort_status": self.abort_signal.is_aborted(),
            "abort_reason": self.abort_signal.get_reason(),
            "completed": completed_flag,
        }

    def get_phase_metrics(self) -> dict[str, Any]:
        """Get metrics for all phases."""
        return {
            str(phase_id): instr.build_metrics()
            for phase_id, instr in self._phase_instrumentation.items()
        }

    async def monitor_progress_async(self, poll_interval: float = 2.0):
        """Monitor progress asynchronously."""
        while True:
            status = self.get_processing_status()
            yield status
            if status["status"] != "running":
                break
            await asyncio.sleep(poll_interval)

    def abort_handler(self, reason: str) -> None:
        """Alias for request_abort."""
        self.request_abort(reason)

    def health_check(self) -> dict[str, Any]:
        """System health check."""
        usage = self.resource_limits.get_resource_usage()
        cpu_headroom = max(
            0.0, self.resource_limits.max_cpu_percent - usage.get("cpu_percent", 0.0)
        )
        mem_headroom = max(
            0.0, (self.resource_limits.max_memory_mb or 0.0) - usage.get("rss_mb", 0.0)
        )

        score = max(
            0.0,
            min(
                100.0,
                (cpu_headroom / max(1.0, self.resource_limits.max_cpu_percent)) * 50.0,
            ),
        )
```

```
8065:
8066:         if self.resource_limits.max_memory_mb:
8067:             score += max(
8068:                 0.0,
8069:                 min(
8070:                     50.0,
8071:                     (mem_headroom / max(1.0, self.resource_limits.max_memory_mb))
8072:                     * 50.0,
8073:                 ),
8074:             )
8075:
8076:         score = min(100.0, score)
8077:
8078:         if self.abort_signal.is_aborted():
8079:             score = min(score, 20.0)
8080:
8081:         return {
8082:             "score": score,
8083:             "resource_usage": usage,
8084:             "abort": self.abort_signal.is_aborted(),
8085:         }
8086:
8087:     def get_system_health(self) -> dict[str, Any]:
8088:         """Comprehensive system health check."""
8089:         health: dict[str, Any] = {
8090:             "status": "healthy",
8091:             "timestamp": datetime.utcnow().isoformat(),
8092:             "components": {},
8093:         }
8094:         components: dict[str, dict[str, Any]] = {}
8095:
8096:         try:
8097:             executor_health: dict[str, Any] = {
8098:                 "instances_loaded": len(self.executor.instances),
8099:                 "status": "healthy",
8100:             }
8101:             components["method_executor"] = executor_health
8102:         except Exception as e:
8103:             health["status"] = "unhealthy"
8104:             components["method_executor"] = {
8105:                 "status": "unhealthy",
8106:                 "error": str(e),
8107:             }
8108:
8109:         try:
8110:             from farfan_pipeline.core.orchestrator.questionnaire import get_questionnaire_provider
8111:             provider = get_questionnaire_provider()
8112:             questionnaire_health: dict[str, Any] = {
8113:                 "has_data": provider.has_data(),
8114:                 "status": "healthy" if provider.has_data() else "unhealthy",
8115:             }
8116:             components["questionnaire_provider"] = questionnaire_health
8117:
8118:             if not provider.has_data():
8119:                 health["status"] = "degraded"
8120:         except Exception as e:
```

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

        try:
            usage = self.resource_limits.get_resource_usage()
            resource_health: dict[str, Any] = {
                "cpu_percent": usage.get("cpu_percent", 0),
                "memory_mb": usage.get("rss_mb", 0),
                "worker_budget": usage.get("worker_budget", 0),
                "status": "healthy",
            }

            if usage.get("cpu_percent", 0) > 80:
                resource_health["status"] = "degraded"
                resource_health["warning"] = "High CPU usage"
                health["status"] = "degraded"

            if usage.get("rss_mb", 0) > 3500:
                resource_health["status"] = "degraded"
                resource_health["warning"] = "High memory usage"
                health["status"] = "degraded"

            components["resources"] = resource_health
        except Exception as e:
            health["status"] = "unhealthy"
            components["resources"] = {"status": "unhealthy", "error": str(e)}

        if self.abort_signal.is_aborted():
            health["status"] = "unhealthy"
            health["abort_reason"] = self.abort_signal.get_reason()

        health["components"] = components
        return health

    def export_metrics(self) -> dict[str, Any]:
        """Export all metrics for monitoring."""
        abort_timestamp = self.abort_signal.get_timestamp()

        return {
            "timestamp": datetime.utcnow().isoformat(),
            "phase_metrics": self.get_phase_metrics(),
            "resource_usage": self.resource_limits.get_usage_history(),
            "abort_status": {
                "is_aborted": self.abort_signal.is_aborted(),
                "reason": self.abort_signal.get_reason(),
                "timestamp": abort_timestamp.isoformat() if abort_timestamp else None,
            },
            "phase_status": dict(self._phase_status),
        }


================================================================================
FILE: src/farfan_pipeline/core/orchestrator/core_clean.py
```

```
8177: ================================================================================
8178:
8179: """F.A.R.F.A.N Orchestrator - Production Version
8180:
8181: 11-phase deterministic policy analysis pipeline with:
8182: - Abort signal propagation
8183: - Adaptive resource management
8184: - Comprehensive instrumentation
8185: - Method dispensary pattern support
8186: - Signal enrichment integration
8187:
8188: Clean architecture. No legacy code. Production-ready.
8189: """
8190:
8191: from __future__ import annotations
8192:
8193: import asyncio
8194: import hashlib
8195: import inspect
8196: import json
8197: import logging
8198: import os
8199: import statistics
8200: import threading
8201: import time
8202: from collections import deque
8203: from dataclasses import dataclass, field, asdict
8204: from datetime import datetime
8205: from pathlib import Path
8206: from types import MappingProxyType
8207: from typing import TYPE_CHECKING, Any, Callable, TypeVar, ParamSpec, TypedDict
8208:
8209: if TYPE_CHECKING:
8210:     from farfan_pipeline.core.orchestrator.factory import CanonicalQuestionnaire
8211:
8212: from farfan_pipeline.core.analysis_port import RecommendationEnginePort
8213: from farfan_pipeline.config.paths import PROJECT_ROOT, RULES_DIR
8214: from farfan_pipeline.processing.aggregation import (
8215:     AggregationSettings,
8216:     AreaPolicyAggregator,
8217:     AreaScore,
8218:     ClusterAggregator,
8219:     ClusterScore,
8220:     DimensionAggregator,
8221:     DimensionScore,
8222:     MacroAggregator,
8223:     MacroScore,
8224:     group_by,
8225:     validate_scored_results,
8226: )
8227: from farfan_pipeline.utils.paths import safe_join
8228: from farfan_pipeline.core.dependency_lockdown import get_dependency_lockdown
8229: from farfan_pipeline.core.types import PreprocessedDocument
8230: from farfan_pipeline.core.orchestrator import executors_contract as executors
8231: from farfan_pipeline.core.orchestrator.arg_router import (
8232:     ArgRouterError,
```

```
8233:     ArgumentValidationError,
8234:     ExtendedArgRouter,
8235: )
8236: from farfan_pipeline.core.orchestrator.class_registry import ClassRegistryError
8237: from farfan_pipeline.core.orchestrator.executor_config import ExecutorConfig
8238: from farfan_pipeline.core.orchestrator.irrigation_synchronizer import (
8239:     IrrigationSynchronizer,
8240:     ExecutionPlan,
8241: )
8242:
8243: logger = logging.getLogger(__name__)
8244: _CORE_MODULE_DIR = Path(__file__).resolve().parent
8245:
8246: EXPECTED_QUESTION_COUNT = int(os.getenv("EXPECTED_QUESTION_COUNT", "305"))
8247: EXPECTED_METHOD_COUNT = int(os.getenv("EXPECTED_METHOD_COUNT", "416"))
8248: PHASE_TIMEOUT_DEFAULT = int(os.getenv("PHASE_TIMEOUT_SECONDS", "300"))
8249: P01_EXPECTED_CHUNK_COUNT = 60
8250: TIMEOUT_SYNC_PHASES: set[int] = {1}
8251:
8252: P = ParamSpec("P")
8253: T = TypeVar("T")
8254:
8255:
8256: # ============================================================================
8257: # PATH RESOLUTION
8258: # ============================================================================
8259:
8260: def resolve_workspace_path(
8261:     path: str | Path,
8262:     *,
8263:     project_root: Path = PROJECT_ROOT,
8264:     rules_dir: Path = RULES_DIR,
8265:     module_dir: Path = _CORE_MODULE_DIR,
8266: ) -> Path:
8267:     """Resolve repository-relative paths deterministically."""
8268:     path_obj = Path(path)
8269:
8270:     if path_obj.is_absolute():
8271:         return path_obj
8272:
8273:     sanitized = safe_join(project_root, *path_obj.parts)
8274:     candidates = [
8275:         sanitized,
8276:         safe_join(module_dir, *path_obj.parts),
8277:         safe_join(rules_dir, *path_obj.parts),
8278:     ]
8279:
8280:     if not path_obj.parts or path_obj.parts[0] != "rules":
8281:         candidates.append(safe_join(rules_dir, "METODOS", *path_obj.parts))
8282:
8283:     for candidate in candidates:
8284:         if candidate.exists():
8285:             return candidate
8286:
8287:     return sanitized
8288:
```

```
8289:
8290: def _normalize_monolith_for_hash(monolith: dict | MappingProxyType) -> dict:
8291:     """Normalize monolith for hash computation."""
8292:     if isinstance(monolith, MappingProxyType):
8293:         monolith = dict(monolith)
8294:
8295:     def _convert(obj: Any) -> Any:
8296:         if isinstance(obj, MappingProxyType):
8297:             obj = dict(obj)
8298:         if isinstance(obj, dict):
8299:             return {k: _convert(v) for k, v in obj.items()}
8300:         if isinstance(obj, list):
8301:             return [_convert(v) for v in obj]
8302:         return obj
8303:
8304:     normalized = _convert(monolith)
8305:
8306:     try:
8307:         json.dumps(normalized, sort_keys=True, ensure_ascii=False, separators=(",", ":"))
8308:     except (TypeError, ValueError) as exc:
8309:         raise RuntimeError(f"Monolith normalization failed: {exc}") from exc
8310:
8311:     return normalized
8312:
8313:
8314: # ==============================================================================
8315: # DATA STRUCTURES
8316: # ==============================================================================
8317:
8318: class MacroScoreDict(TypedDict):
8319:     """Typed container for macro score results."""
8320:     macro_score: MacroScore
8321:     macro_score_normalized: float
8322:     cluster_scores: list[ClusterScore]
8323:     cross_cutting_coherence: float
8324:     systemic_gaps: list[str]
8325:     strategic_alignment: float
8326:     quality_band: str
8327:
8328:
8329: @dataclass
8330: class ClusterScoreData:
8331:     """Cluster score data."""
8332:     id: str
8333:     score: float
8334:     normalized_score: float
8335:
8336:
8337: @dataclass
8338: class MacroEvaluation:
8339:     """Macro evaluation result."""
8340:     macro_score: float
8341:     macro_score_normalized: float
8342:     clusters: list[ClusterScoreData]
8343:
8344:
```

```
@dataclass
class Evidence:
    """Evidence container."""
    modality: str
    elements: list[Any] = field(default_factory=list)
    raw_results: dict[str, Any] = field(default_factory=dict)


@dataclass
class PhaseResult:
    """Phase execution result."""
    success: bool
    phase_id: str
    data: Any
    error: Exception | None
    duration_ms: float
    mode: str
    aborted: bool = False


@dataclass
class MicroQuestionRun:
    """Micro-question execution result."""
    question_id: str
    question_global: int
    base_slot: str
    metadata: dict[str, Any]
    evidence: Evidence | None
    error: str | None = None
    duration_ms: float | None = None
    aborted: bool = False


@dataclass
class ScoredMicroQuestion:
    """Scored micro-question."""
    question_id: str
    question_global: int
    base_slot: str
    score: float | None
    normalized_score: float | None
    quality_level: str | None
    evidence: Evidence | None
    scoring_details: dict[str, Any] = field(default_factory=dict)
    metadata: dict[str, Any] = field(default_factory=dict)
    error: str | None = None


# ============================================================================
# ABORT MECHANISM
# ============================================================================

class AbortRequested(RuntimeError):
    """Abort signal exception."""
    pass

```

```
8401:
8402: class AbortSignal:
8403:     """Thread-safe abort signal."""
8404:
8405:     def __init__(self) -> None:
8406:         self._event = threading.Event()
8407:         self._lock = threading.Lock()
8408:         self._reason: str | None = None
8409:         self._timestamp: datetime | None = None
8410:
8411:     def abort(self, reason: str) -> None:
8412:         """Trigger abort."""
8413:         if not reason:
8414:             reason = "Abort requested"
8415:         with self._lock:
8416:             if not self._event.is_set():
8417:                 self._event.set()
8418:                 self._reason = reason
8419:                 self._timestamp = datetime.utcnow()
8420:
8421:     def is_aborted(self) -> bool:
8422:         """Check if aborted."""
8423:         return self._event.is_set()
8424:
8425:     def get_reason(self) -> str | None:
8426:         """Get abort reason."""
8427:         with self._lock:
8428:             return self._reason
8429:
8430:     def get_timestamp(self) -> datetime | None:
8431:         """Get abort timestamp."""
8432:         with self._lock:
8433:             return self._timestamp
8434:
8435:     def reset(self) -> None:
8436:         """Reset abort signal."""
8437:         with self._lock:
8438:             self._event.clear()
8439:             self._reason = None
8440:             self._timestamp = None
8441:
8442:
8443: # ============================================================================
8444: # RESOURCE MANAGEMENT
8445: # ============================================================================
8446:
8447: class ResourceLimits:
8448:     """Adaptive resource management."""
8449:
8450:     def __init__(
8451:         self,
8452:         max_memory_mb: float | None = 4096.0,
8453:         max_cpu_percent: float = 85.0,
8454:         max_workers: int = 32,
8455:         min_workers: int = 4,
8456:         hard_max_workers: int = 64,
```

```
8457:         history: int = 120,
8458:     ) -> None:
8459:         self.max_memory_mb = max_memory_mb
8460:         self.max_cpu_percent = max_cpu_percent
8461:         self.min_workers = max(1, min_workers)
8462:         self.hard_max_workers = max(self.min_workers, hard_max_workers)
8463:         self._max_workers = max(self.min_workers, min(max_workers, self.hard_max_workers))
8464:         self._usage_history: deque[dict[str, float]] = deque(maxlen=history)
8465:         self._semaphore: asyncio.Semaphore | None = None
8466:         self._semaphore_limit = self._max_workers
8467:         self._async_lock: asyncio.Lock | None = None
8468:         self._psutil = None
8469:         self._psutil_process = None
8470:
8471:         try:
8472:             import psutil
8473:             self._psutil = psutil
8474:             self._psutil_process = psutil.Process(os.getpid())
8475:         except Exception:
8476:             logger.warning("psutil unavailable, using fallbacks")
8477:
8478:     @property
8479:     def max_workers(self) -> int:
8480:         return self._max_workers
8481:
8482:     def attach_semaphore(self, semaphore: asyncio.Semaphore) -> None:
8483:         """Attach semaphore for budget control."""
8484:         self._semaphore = semaphore
8485:         self._semaphore_limit = self._max_workers
8486:
8487:     async def apply_worker_budget(self) -> int:
8488:         """Apply worker budget to semaphore."""
8489:         if self._semaphore is None:
8490:             return self._max_workers
8491:
8492:         if self._async_lock is None:
8493:             self._async_lock = asyncio.Lock()
8494:
8495:         async with self._async_lock:
8496:             desired = self._max_workers
8497:             current = self._semaphore_limit
8498:
8499:             if desired > current:
8500:                 for _ in range(desired - current):
8501:                     self._semaphore.release()
8502:             elif desired < current:
8503:                 for _ in range(current - desired):
8504:                     await self._semaphore.acquire()
8505:
8506:             self._semaphore_limit = desired
8507:             return self._max_workers
8508:
8509:     def _record_usage(self, usage: dict[str, float]) -> None:
8510:         """Record usage and predict budget."""
8511:         self._usage_history.append(usage)
8512:         self._predict_worker_budget()
```

```
8513:
8514:     def _predict_worker_budget(self) -> None:
8515:         """Adaptive worker budget prediction."""
8516:         if len(self._usage_history) < 5:
8517:             return
8518:
8519:         recent_cpu = [e["cpu_percent"] for e in list(self._usage_history)[-5:]]
8520:         recent_mem = [e["memory_percent"] for e in list(self._usage_history)[-5:]]
8521:
8522:         avg_cpu = statistics.mean(recent_cpu)
8523:         avg_mem = statistics.mean(recent_mem)
8524:
8525:         new_budget = self._max_workers
8526:
8527:         if (self.max_cpu_percent and avg_cpu > self.max_cpu_percent * 0.95) or \
8528:            (self.max_memory_mb and avg_mem > 90.0):
8529:             new_budget = max(self.min_workers, self._max_workers - 1)
8530:         elif avg_cpu < self.max_cpu_percent * 0.6 and avg_mem < 70.0:
8531:             new_budget = min(self.hard_max_workers, self._max_workers + 1)
8532:
8533:         self._max_workers = max(self.min_workers, min(new_budget, self.hard_max_workers))
8534:
8535:     def get_resource_usage(self) -> dict[str, float]:
8536:         """Get current resource usage."""
8537:         timestamp = datetime.utcnow().isoformat()
8538:         cpu_percent = 0.0
8539:         memory_percent = 0.0
8540:         rss_mb = 0.0
8541:
8542:         if self._psutil:
8543:             try:
8544:                 cpu_percent = float(self._psutil.cpu_percent(interval=None))
8545:                 virtual_memory = self._psutil.virtual_memory()
8546:                 memory_percent = float(virtual_memory.percent)
8547:                 if self._psutil_process:
8548:                     rss_mb = float(self._psutil_process.memory_info().rss / (1024 * 1024))
8549:             except Exception:
8550:                 cpu_percent = 0.0
8551:         else:
8552:             try:
8553:                 load1, _, _ = os.getloadavg()
8554:                 cpu_percent = float(min(100.0, load1 * 100))
8555:             except OSError:
8556:                 cpu_percent = 0.0
8557:
8558:             try:
8559:                 import resource
8560:                 usage_info = resource.getrusage(resource.RUSAGE_SELF)
8561:                 rss_mb = float(usage_info.ru_maxrss / 1024)
8562:             except Exception:
8563:                 rss_mb = 0.0
8564:
8565:         usage = {
8566:             "timestamp": timestamp,
8567:             "cpu_percent": cpu_percent,
8568:             "memory_percent": memory_percent,
```

```
            "rss_mb": rss_mb,
            "worker_budget": float(self._max_workers),
        }

        self._record_usage(usage)
        return usage

    def check_memory_exceeded(self, usage: dict[str, float] | None = None) -> tuple[bool, dict[str, float]]:
        """Check memory limit."""
        usage = usage or self.get_resource_usage()
        exceeded = False
        if self.max_memory_mb is not None:
            exceeded = usage.get("rss_mb", 0.0) > self.max_memory_mb
        return exceeded, usage

    def check_cpu_exceeded(self, usage: dict[str, float] | None = None) -> tuple[bool, dict[str, float]]:
        """Check CPU limit."""
        usage = usage or self.get_resource_usage()
        exceeded = False
        if self.max_cpu_percent:
            exceeded = usage.get("cpu_percent", 0.0) > self.max_cpu_percent
        return exceeded, usage

    def get_usage_history(self) -> list[dict[str, float]]:
        """Get usage history."""
        return list(self._usage_history)


# ============================================================================
# INSTRUMENTATION
# ============================================================================

class PhaseInstrumentation:
    """Phase telemetry collection."""

    def __init__(
        self,
        phase_id: int,
        name: str,
        items_total: int | None = None,
        snapshot_interval: int = 10,
        resource_limits: ResourceLimits | None = None,
    ) -> None:
        self.phase_id = phase_id
        self.name = name
        self.items_total = items_total or 0
        self.snapshot_interval = max(1, snapshot_interval)
        self.resource_limits = resource_limits
        self.items_processed = 0
        self.start_time: float | None = None
        self.end_time: float | None = None
        self.warnings: list[dict[str, Any]] = []
        self.errors: list[dict[str, Any]] = []
        self.resource_snapshots: list[dict[str, Any]] = []
        self.latencies: list[float] = []
        self.anomalies: list[dict[str, Any]] = []
```

```
8625:
8626:     def start(self, items_total: int | None = None) -> None:
8627:         """Start phase."""
8628:         if items_total is not None:
8629:             self.items_total = items_total
8630:         self.start_time = time.perf_counter()
8631:
8632:     def increment(self, count: int = 1, latency: float | None = None) -> None:
8633:         """Increment progress."""
8634:         self.items_processed += count
8635:         if latency is not None:
8636:             self.latencies.append(latency)
8637:             self._detect_latency_anomaly(latency)
8638:         if self.resource_limits and self.should_snapshot():
8639:             self.capture_resource_snapshot()
8640:
8641:     def should_snapshot(self) -> bool:
8642:         """Check if snapshot needed."""
8643:         if self.items_total == 0 or self.items_processed == 0:
8644:             return False
8645:         return self.items_processed % self.snapshot_interval == 0
8646:
8647:     def capture_resource_snapshot(self) -> None:
8648:         """Capture resource snapshot."""
8649:         if not self.resource_limits:
8650:             return
8651:         snapshot = self.resource_limits.get_resource_usage()
8652:         snapshot["items_processed"] = self.items_processed
8653:         self.resource_snapshots.append(snapshot)
8654:
8655:     def record_warning(self, category: str, message: str, **extra: Any) -> None:
8656:         """Record warning."""
8657:         entry = {
8658:             "category": category,
8659:             "message": message,
8660:             **extra,
8661:             "timestamp": datetime.utcnow().isoformat(),
8662:         }
8663:         self.warnings.append(entry)
8664:
8665:     def record_error(self, category: str, message: str, **extra: Any) -> None:
8666:         """Record error."""
8667:         entry = {
8668:             "category": category,
8669:             "message": message,
8670:             **extra,
8671:             "timestamp": datetime.utcnow().isoformat(),
8672:         }
8673:         self.errors.append(entry)
8674:
8675:     def _detect_latency_anomaly(self, latency: float) -> None:
8676:         """Detect latency spikes."""
8677:         if len(self.latencies) < 5:
8678:             return
8679:
8680:         mean_latency = statistics.mean(self.latencies)
```

```
        std_latency = statistics.pstdev(self.latencies) or 0.0
        threshold = mean_latency + (3 * std_latency)

        if std_latency and latency > threshold:
            self.anomalies.append({
                "type": "latency_spike",
                "latency": latency,
                "mean": mean_latency,
                "std": std_latency,
                "timestamp": datetime.utcnow().isoformat(),
            })

    def complete(self) -> None:
        """Complete phase."""
        self.end_time = time.perf_counter()

    def duration_ms(self) -> float | None:
        """Get duration."""
        if self.start_time is None or self.end_time is None:
            return None
        return (self.end_time - self.start_time) * 1000.0

    def progress(self) -> float | None:
        """Get progress fraction."""
        if not self.items_total:
            return None
        return min(1.0, self.items_processed / float(self.items_total))

    def throughput(self) -> float | None:
        """Get items per second."""
        if self.start_time is None:
            return None
        elapsed = (time.perf_counter() - self.start_time) if self.end_time is None else (self.end_time - self.start_time)
        if not elapsed:
            return None
        return self.items_processed / elapsed

    def latency_histogram(self) -> dict[str, float | None]:
        """Get latency percentiles."""
        if not self.latencies:
            return {"p50": None, "p95": None, "p99": None}

        sorted_latencies = sorted(self.latencies)

        def percentile(p: float) -> float:
            if not sorted_latencies:
                return 0.0
            k = (len(sorted_latencies) - 1) * (p / 100.0)
            f = int(k)
            c = min(f + 1, len(sorted_latencies) - 1)
            if f == c:
                return sorted_latencies[int(k)]
            d0 = sorted_latencies[f] * (c - k)
            d1 = sorted_latencies[c] * (k - f)
            return d0 + d1

```

```
8737:         return {
8738:             "p50": percentile(50.0),
8739:             "p95": percentile(95.0),
8740:             "p99": percentile(99.0),
8741:         }
8742:
8743:     def build_metrics(self) -> dict[str, Any]:
8744:         """Build metrics summary."""
8745:         return {
8746:             "phase_id": self.phase_id,
8747:             "name": self.name,
8748:             "duration_ms": self.duration_ms(),
8749:             "items_processed": self.items_processed,
8750:             "items_total": self.items_total,
8751:             "progress": self.progress(),
8752:             "throughput": self.throughput(),
8753:             "warnings": list(self.warnings),
8754:             "errors": list(self.errors),
8755:             "resource_snapshots": list(self.resource_snapshots),
8756:             "latency_histogram": self.latency_histogram(),
8757:             "anomalies": list(self.anomalies),
8758:         }
8759:
8760:
8761: # ============================================================================
8762: # TIMEOUT HANDLING
8763: # ============================================================================
8764:
8765: class PhaseTimeoutError(RuntimeError):
8766:     """Phase timeout exception."""
8767:
8768:     def __init__(self, phase_id: int | str, phase_name: str, timeout_s: float) -> None:
8769:         self.phase_id = phase_id
8770:         self.phase_name = phase_name
8771:         self.timeout_s = timeout_s
8772:         super().__init__(f"Phase {phase_id} ({phase_name}) timed out after {timeout_s}s")
8773:
8774:
8775: async def execute_phase_with_timeout(
8776:     phase_id: int,
8777:     phase_name: str,
8778:     coro: Callable[P, T] | None = None,
8779:     handler: Callable[P, T] | None = None,
8780:     args: tuple | None = None,
8781:     timeout_s: float = 300.0,
8782:     **kwargs: P.kwargs,
8783: ) -> T:
8784:     """Execute phase with timeout."""
8785:     target = coro or handler
8786:     if target is None:
8787:         raise ValueError("Either 'coro' or 'handler' must be provided")
8788:
8789:     call_args = args or ()
8790:
8791:     start = time.perf_counter()
8792:     logger.info(f"Phase {phase_id} ({phase_name}) started, timeout={timeout_s}s")
```

```
8793:
8794:     try:
8795:         result = await asyncio.wait_for(target(*call_args, **kwargs), timeout=timeout_s)
8796:         elapsed = time.perf_counter() - start
8797:         logger.info(f"Phase {phase_id} completed in {elapsed:.2f}s")
8798:         return result
8799:
8800:     except asyncio.TimeoutError as exc:
8801:         elapsed = time.perf_counter() - start
8802:         logger.error(f"Phase {phase_id} TIMEOUT after {elapsed:.2f}s")
8803:         raise PhaseTimeoutError(phase_id, phase_name, timeout_s) from exc
8804:
8805:     except asyncio.CancelledError:
8806:         elapsed = time.perf_counter() - start
8807:         logger.warning(f"Phase {phase_id} CANCELLED after {elapsed:.2f}s")
8808:         raise
8809:
8810:     except Exception as exc:
8811:         elapsed = time.perf_counter() - start
8812:         logger.error(f"Phase {phase_id} ERROR: {exc} (after {elapsed:.2f}s)", exc_info=True)
8813:         raise
8814:
8815:
8816: # ============================================================================
8817: # METHOD EXECUTOR
8818: # ============================================================================
8819:
8820: class _LazyInstanceDict:
8821:     """Lazy instance dictionary."""
8822:
8823:     def __init__(self, method_registry: Any) -> None:
8824:         self._registry = method_registry
8825:
8826:     def get(self, class_name: str, default: Any = None) -> Any:
8827:         try:
8828:             return self._registry._get_instance(class_name)
8829:         except Exception:
8830:             return default
8831:
8832:     def __getitem__(self, class_name: str) -> Any:
8833:         return self._registry._get_instance(class_name)
8834:
8835:     def __contains__(self, class_name: str) -> bool:
8836:         return class_name in self._registry._class_paths
8837:
8838:     def keys(self) -> list[str]:
8839:         return list(self._registry._class_paths.keys())
8840:
8841:     def values(self) -> list[Any]:
8842:         return [self.get(name) for name in self.keys()]
8843:
8844:     def items(self) -> list[tuple[str, Any]]:
8845:         return [(name, self.get(name)) for name in self.keys()]
8846:
8847:     def __len__(self) -> int:
8848:         return len(self._registry._class_paths)
```

```
8849:
8850:
8851: class MethodExecutor:
8852:     """Method executor with lazy loading."""
8853:
8854:     def __init__(
8855:         self,
8856:         dispatcher: Any | None = None,
8857:         signal_registry: Any | None = None,
8858:         method_registry: Any | None = None,
8859:     ) -> None:
8860:         from farfan_pipeline.core.orchestrator.method_registry import (
8861:             MethodRegistry,
8862:             setup_default_instantiation_rules,
8863:         )
8864:
8865:         self.degraded_mode = False
8866:         self.degraded_reasons: list[str] = []
8867:         self.signal_registry = signal_registry
8868:
8869:         if method_registry is not None:
8870:             self._method_registry = method_registry
8871:         else:
8872:             try:
8873:                 self._method_registry = MethodRegistry()
8874:                 setup_default_instantiation_rules(self._method_registry)
8875:                 logger.info("Method registry initialized")
8876:             except Exception as exc:
8877:                 self.degraded_mode = True
8878:                 reason = f"Method registry initialization failed: {exc}"
8879:                 self.degraded_reasons.append(reason)
8880:                 logger.error(f"DEGRADED MODE: {reason}")
8881:                 self._method_registry = MethodRegistry(class_paths={})
8882:
8883:         try:
8884:             from farfan_pipeline.core.orchestrator.class_registry import build_class_registry
8885:             registry = build_class_registry()
8886:         except (ClassRegistryError, ModuleNotFoundError, ImportError) as exc:
8887:             self.degraded_mode = True
8888:             reason = f"Could not build class registry: {exc}"
8889:             self.degraded_reasons.append(reason)
8890:             logger.warning(f"DEGRADED MODE: {reason}")
8891:             registry = {}
8892:
8893:         self._router = ExtendedArgRouter(registry)
8894:         self.instances = _LazyInstanceDict(self._method_registry)
8895:
8896:     @staticmethod
8897:     def _supports_parameter(callable_obj: Any, parameter_name: str) -> bool:
8898:         try:
8899:             signature = inspect.signature(callable_obj)
8900:         except (TypeError, ValueError):
8901:             return False
8902:         return parameter_name in signature.parameters
8903:
8904:     def execute(self, class_name: str, method_name: str, **kwargs: Any) -> Any:
```

```
8905:         """Execute method."""
8906:         from farfan_pipeline.core.orchestrator.method_registry import MethodRegistryError
8907:
8908:         try:
8909:             method = self._method_registry.get_method(class_name, method_name)
8910:         except MethodRegistryError as exc:
8911:             logger.error(f"Method retrieval failed: {class_name}.{method_name}: {exc}")
8912:             if self.degraded_mode:
8913:                 logger.warning("Returning None due to degraded mode")
8914:                 return None
8915:             raise AttributeError(f"Cannot retrieve {class_name}.{method_name}: {exc}") from exc
8916:
8917:         try:
8918:             args, routed_kwargs = self._router.route(class_name, method_name, dict(kwargs))
8919:             return method(*args, **routed_kwargs)
8920:         except (ArgRouterError, ArgumentValidationError):
8921:             logger.exception(f"Argument routing failed for {class_name}.{method_name}")
8922:             raise
8923:         except Exception:
8924:             logger.exception(f"Method execution failed for {class_name}.{method_name}")
8925:             raise
8926:
8927:     def inject_method(self, class_name: str, method_name: str, method: Callable[..., Any]) -> None:
8928:         """Inject method."""
8929:         self._method_registry.inject_method(class_name, method_name, method)
8930:         logger.info(f"Method injected: {class_name}.{method_name}")
8931:
8932:     def has_method(self, class_name: str, method_name: str) -> bool:
8933:         """Check if method exists."""
8934:         return self._method_registry.has_method(class_name, method_name)
8935:
8936:     def get_registry_stats(self) -> dict[str, Any]:
8937:         """Get registry stats."""
8938:         return self._method_registry.get_stats()
8939:
8940:     def get_routing_metrics(self) -> dict[str, Any]:
8941:         """Get routing metrics."""
8942:         if hasattr(self._router, "get_metrics"):
8943:             return self._router.get_metrics()
8944:         return {}
8945:
8946:
8947: # ============================================================================
8948: # PHASE VALIDATION
8949: # ============================================================================
8950:
8951: def validate_phase_definitions(
8952:     phase_list: list[tuple[int, str, str, str]], orchestrator_class: type
8953: ) -> None:
8954:     """Validate phase definitions."""
8955:     if not phase_list:
8956:         raise RuntimeError("FASES cannot be empty")
8957:
8958:     phase_ids = [phase[0] for phase in phase_list]
8959:
8960:     seen_ids = set()
```

```
8961:     for phase_id in phase_ids:
8962:         if phase_id in seen_ids:
8963:             raise RuntimeError(f"Duplicate phase ID {phase_id}")
8964:         seen_ids.add(phase_id)
8965:
8966:     if phase_ids != sorted(phase_ids):
8967:         raise RuntimeError(f"Phase IDs must be sorted. Got {phase_ids}")
8968:     if phase_ids[0] != 0:
8969:         raise RuntimeError(f"Phase IDs must start from 0. Got {phase_ids[0]}")
8970:     if phase_ids[-1] != len(phase_list) - 1:
8971:         raise RuntimeError(f"Phase IDs must be contiguous. Got {phase_ids[-1]}")
8972:
8973:     valid_modes = {"sync", "async"}
8974:     for phase_id, mode, handler_name, label in phase_list:
8975:         if mode not in valid_modes:
8976:             raise RuntimeError(f"Phase {phase_id}: invalid mode '{mode}'")
8977:
8978:         if not hasattr(orchestrator_class, handler_name):
8979:             raise RuntimeError(f"Phase {phase_id}: missing handler '{handler_name}'")
8980:
8981:         handler = getattr(orchestrator_class, handler_name, None)
8982:         if not callable(handler):
8983:             raise RuntimeError(f"Phase {phase_id}: handler '{handler_name}' not callable")
8984:
8985:
8986: # ============================================================================
8987: # ORCHESTRATOR
8988: # ============================================================================
8989:
8990: class Orchestrator:
8991:     """11-phase deterministic policy analysis orchestrator."""
8992:
8993:     FASES: list[tuple[int, str, str, str]] = [
8994:         (0, "sync", "_load_configuration", "FASE 0 - ConfiguraciÃ³n"),
8995:         (1, "sync", "_ingest_document", "FASE 1 - IngestiÃ³n"),
8996:         (2, "async", "_execute_micro_questions_async", "FASE 2 - Micro Preguntas"),
8997:         (3, "async", "_score_micro_results_async", "FASE 3 - Scoring"),
8998:         (4, "async", "_aggregate_dimensions_async", "FASE 4 - Dimensiones"),
8999:         (5, "async", "_aggregate_policy_areas_async", "FASE 5 - Ã\201reas"),
9000:         (6, "sync", "_aggregate_clusters", "FASE 6 - ClÃºsteres"),
9001:         (7, "sync", "_evaluate_macro", "FASE 7 - Macro"),
9002:         (8, "async", "_generate_recommendations", "FASE 8 - Recomendaciones"),
9003:         (9, "sync", "_assemble_report", "FASE 9 - Reporte"),
9004:         (10, "async", "_format_and_export", "FASE 10 - ExportaciÃ³n"),
9005:     ]
9006:
9007:     PHASE_ITEM_TARGETS: dict[int, int] = {
9008:         0: 1, 1: 1, 2: 300, 3: 300, 4: 60,
9009:         5: 10, 6: 4, 7: 1, 8: 1, 9: 1, 10: 1,
9010:     }
9011:
9012:     PHASE_OUTPUT_KEYS: dict[int, str] = {
9013:         0: "config", 1: "document", 2: "micro_results",
9014:         3: "scored_results", 4: "dimension_scores",
9015:         5: "policy_area_scores", 6: "cluster_scores",
9016:         7: "macro_result", 8: "recommendations",
```

```
9017:         9: "report", 10: "export_payload",
9018:     }
9019:
9020:     PHASE_ARGUMENT_KEYS: dict[int, list[str]] = {
9021:         1: ["pdf_path", "config"],
9022:         2: ["document", "config"],
9023:         3: ["micro_results", "config"],
9024:         4: ["scored_results", "config"],
9025:         5: ["dimension_scores", "config"],
9026:         6: ["policy_area_scores", "config"],
9027:         7: ["cluster_scores", "config"],
9028:         8: ["macro_result", "config"],
9029:         9: ["recommendations", "config"],
9030:         10: ["report", "config"],
9031:     }
9032:
9033:     PHASE_TIMEOUTS: dict[int, float] = {
9034:         0: 60, 1: 120, 2: 600, 3: 300, 4: 180,
9035:         5: 120, 6: 60, 7: 60, 8: 120, 9: 60, 10: 120,
9036:     }
9037:
9038:     def __init__(
9039:         self,
9040:         method_executor: MethodExecutor,
9041:         questionnaire: CanonicalQuestionnaire,
9042:         executor_config: ExecutorConfig,
9043:         calibration_orchestrator: Any | None = None,
9044:         resource_limits: ResourceLimits | None = None,
9045:         resource_snapshot_interval: int = 10,
9046:         recommendation_engine_port: RecommendationEnginePort | None = None,
9047:         processor_bundle: Any | None = None,
9048:     ) -> None:
9049:         """Initialize orchestrator."""
9050:         from farfan_pipeline.core.orchestrator.factory import _validate_questionnaire_structure
9051:         from farfan_pipeline.core.orchestrator.questionnaire import get_questionnaire_provider
9052:
9053:         validate_phase_definitions(self.FASES, self.__class__)
9054:
9055:         self.executor = method_executor
9056:         self._canonical_questionnaire = questionnaire
9057:         self._monolith_data = dict(questionnaire.data)
9058:         self.executor_config = executor_config
9059:         self.calibration_orchestrator = calibration_orchestrator
9060:         self.resource_limits = resource_limits or ResourceLimits()
9061:         self.resource_snapshot_interval = max(1, resource_snapshot_interval)
9062:         self.questionnaire_provider = get_questionnaire_provider()
9063:
9064:         self._enriched_packs = None
9065:         if processor_bundle is not None:
9066:             if hasattr(processor_bundle, "enriched_signal_packs"):
9067:                 self._enriched_packs = processor_bundle.enriched_signal_packs
9068:                 logger.info(f"Orchestrator wired with {len(self._enriched_packs)} enriched signal packs")
9069:             else:
9070:                 logger.warning("ProcessorBundle missing enriched_signal_packs")
9071:         else:
9072:             logger.warning("No ProcessorBundle provided")
```

```
9073:
9074:         if not hasattr(self.executor, "signal_registry") or self.executor.signal_registry is None:
9075:             raise RuntimeError("MethodExecutor must have signal_registry")
9076:
9077:         try:
9078:             _validate_questionnaire_structure(self._monolith_data)
9079:         except (ValueError, TypeError) as e:
9080:             raise RuntimeError(f"Questionnaire validation failed: {e}") from e
9081:
9082:         if not self.executor.instances:
9083:             raise RuntimeError("MethodExecutor.instances is empty")
9084:
9085:         self.executors = {
9086:             "D1-Q1": executors.D1Q1_Executor, "D1-Q2": executors.D1Q2_Executor,
9087:             "D1-Q3": executors.D1Q3_Executor, "D1-Q4": executors.D1Q4_Executor,
9088:             "D1-Q5": executors.D1Q5_Executor, "D2-Q1": executors.D2Q1_Executor,
9089:             "D2-Q2": executors.D2Q2_Executor, "D2-Q3": executors.D2Q3_Executor,
9090:             "D2-Q4": executors.D2Q4_Executor, "D2-Q5": executors.D2Q5_Executor,
9091:             "D3-Q1": executors.D3Q1_Executor, "D3-Q2": executors.D3Q2_Executor,
9092:             "D3-Q3": executors.D3Q3_Executor, "D3-Q4": executors.D3Q4_Executor,
9093:             "D3-Q5": executors.D3Q5_Executor, "D4-Q1": executors.D4Q1_Executor,
9094:             "D4-Q2": executors.D4Q2_Executor, "D4-Q3": executors.D4Q3_Executor,
9095:             "D4-Q4": executors.D4Q4_Executor, "D4-Q5": executors.D4Q5_Executor,
9096:             "D5-Q1": executors.D5Q1_Executor, "D5-Q2": executors.D5Q2_Executor,
9097:             "D5-Q3": executors.D5Q3_Executor, "D5-Q4": executors.D5Q4_Executor,
9098:             "D5-Q5": executors.D5Q5_Executor, "D6-Q1": executors.D6Q1_Executor,
9099:             "D6-Q2": executors.D6Q2_Executor, "D6-Q3": executors.D6Q3_Executor,
9100:             "D6-Q4": executors.D6Q4_Executor, "D6-Q5": executors.D6Q5_Executor,
9101:         }
9102:
9103:         self.abort_signal = AbortSignal()
9104:         self.phase_results: list[PhaseResult] = []
9105:         self._phase_instrumentation: dict[int, PhaseInstrumentation] = {}
9106:         self._phase_status: dict[int, str] = {phase_id: "not_started" for phase_id, *_ in self.FASES}
9107:         self._phase_outputs: dict[int, Any] = {}
9108:         self._context: dict[str, Any] = {}
9109:         self._start_time: float | None = None
9110:         self._execution_plan: ExecutionPlan | None = None
9111:
9112:         self.dependency_lockdown = get_dependency_lockdown()
9113:         logger.info(f"Orchestrator initialized: {self.dependency_lockdown.get_mode_description()}")
9114:
9115:         self.recommendation_engine = recommendation_engine_port
9116:         if self.recommendation_engine:
9117:             logger.info("RecommendationEngine port injected")
9118:
9119:     def _ensure_not_aborted(self) -> None:
9120:         """Check abort status."""
9121:         if self.abort_signal.is_aborted():
9122:             reason = self.abort_signal.get_reason() or "Unknown"
9123:             raise AbortRequested(f"Orchestration aborted: {reason}")
9124:
9125:     def request_abort(self, reason: str) -> None:
9126:         """Request abort."""
9127:         self.abort_signal.abort(reason)
9128:
```

```
    def reset_abort(self) -> None:
        """Reset abort signal."""
        self.abort_signal.reset()

    def _get_phase_timeout(self, phase_id: int) -> float:
        """Get phase timeout."""
        return self.PHASE_TIMEOUTS.get(phase_id, 300.0)

    async def process_development_plan_async(
        self, pdf_path: str, preprocessed_document: Any | None = None
    ) -> list[PhaseResult]:
        """Execute 11-phase pipeline."""
        self.reset_abort()
        self.phase_results = []
        self._phase_instrumentation = {}
        self._phase_outputs = {}
        self._context = {"pdf_path": pdf_path}

        if preprocessed_document is not None:
            self._context["preprocessed_override"] = preprocessed_document

        self._phase_status = {phase_id: "not_started" for phase_id, *_ in self.FASES}
        self._start_time = time.perf_counter()

        for phase_id, mode, handler_name, phase_label in self.FASES:
            self._ensure_not_aborted()

            handler = getattr(self, handler_name)
            instrumentation = PhaseInstrumentation(
                phase_id=phase_id,
                name=phase_label,
                items_total=self.PHASE_ITEM_TARGETS.get(phase_id),
                snapshot_interval=self.resource_snapshot_interval,
                resource_limits=self.resource_limits,
            )

            instrumentation.start(items_total=self.PHASE_ITEM_TARGETS.get(phase_id))
            self._phase_instrumentation[phase_id] = instrumentation
            self._phase_status[phase_id] = "running"

            args = [self._context[key] for key in self.PHASE_ARGUMENT_KEYS.get(phase_id, [])]

            success = False
            data: Any = None
            error: Exception | None = None

            try:
                if mode == "sync":
                    if phase_id in TIMEOUT_SYNC_PHASES:
                        data = await execute_phase_with_timeout(
                            phase_id=phase_id,
                            phase_name=phase_label,
                            timeout_s=self._get_phase_timeout(phase_id),
                            coro=asyncio.to_thread,
                            args=(handler,) + tuple(args),
                        )
```

```
                    else:
                        data = handler(*args)
                else:
                    data = await execute_phase_with_timeout(
                        phase_id=phase_id,
                        phase_name=phase_label,
                        timeout_s=self._get_phase_timeout(phase_id),
                        handler=handler,
                        args=tuple(args),
                    )
                success = True

            except PhaseTimeoutError as exc:
                error = exc
                instrumentation.record_error("timeout", str(exc))
                self.request_abort(f"Phase {phase_id} timed out")

            except AbortRequested as exc:
                error = exc
                instrumentation.record_warning("abort", str(exc))

            except Exception as exc:
                logger.exception(f"Phase {phase_label} failed")
                error = exc
                instrumentation.record_error("exception", str(exc))
                self.request_abort(f"Phase {phase_id} failed: {exc}")

            finally:
                instrumentation.complete()

            aborted = self.abort_signal.is_aborted()
            duration_ms = instrumentation.duration_ms() or 0.0

            phase_result = PhaseResult(
                success=success and not aborted,
                phase_id=str(phase_id),
                data=data,
                error=error,
                duration_ms=duration_ms,
                mode=mode,
                aborted=aborted,
            )
            self.phase_results.append(phase_result)

            if success and not aborted:
                self._phase_outputs[phase_id] = data
                out_key = self.PHASE_OUTPUT_KEYS.get(phase_id)
                if out_key:
                    self._context[out_key] = data
                self._phase_status[phase_id] = "completed"

                if phase_id == 1:
                    try:
                        document = self._context.get("document")
                        chunks = getattr(document, "chunks", []) if document else []

```

```
                        synchronizer = IrrigationSynchronizer(
                            questionnaire=self._monolith_data,
                            document_chunks=chunks
                        )
                        self._execution_plan = synchronizer.build_execution_plan()

                        logger.info(f"Execution plan built: {len(self._execution_plan.tasks)} tasks")
                    except ValueError as e:
                        logger.error(f"Failed to build execution plan: {e}")
                        self.request_abort(f"Synchronization failed: {e}")
                        raise
            elif aborted:
                self._phase_status[phase_id] = "aborted"
                break
            else:
                self._phase_status[phase_id] = "failed"
                break

        return self.phase_results

    def process_development_plan(
        self, pdf_path: str, preprocessed_document: Any | None = None
    ) -> list[PhaseResult]:
        """Sync wrapper."""
        try:
            loop = asyncio.get_running_loop()
        except RuntimeError:
            loop = None

        if loop and loop.is_running():
            raise RuntimeError("Cannot call from within async context")

        return asyncio.run(self.process_development_plan_async(pdf_path, preprocessed_document))

    def get_processing_status(self) -> dict[str, Any]:
        """Get status."""
        if self._start_time is None:
            status = "not_started"
            elapsed = 0.0
            completed_flag = False
        else:
            aborted = self.abort_signal.is_aborted()
            status = "aborted" if aborted else "running"
            elapsed = time.perf_counter() - self._start_time
            completed_flag = all(state == "completed" for state in self._phase_status.values()) and not aborted

        completed = sum(1 for state in self._phase_status.values() if state == "completed")
        total = len(self.FASES)
        overall_progress = completed / total if total else 0.0

        phase_progress = {
            str(phase_id): instr.progress()
            for phase_id, instr in self._phase_instrumentation.items()
        }

        resource_usage = self.resource_limits.get_resource_usage() if self._start_time else {}
```

```
9297:
9298:         return {
9299:             "status": status,
9300:             "overall_progress": overall_progress,
9301:             "phase_progress": phase_progress,
9302:             "elapsed_time_s": elapsed,
9303:             "resource_usage": resource_usage,
9304:             "abort_status": self.abort_signal.is_aborted(),
9305:             "abort_reason": self.abort_signal.get_reason(),
9306:             "completed": completed_flag,
9307:         }
9308:
9309:     def get_phase_metrics(self) -> dict[str, Any]:
9310:         """Get phase metrics."""
9311:         return {
9312:             str(phase_id): instr.build_metrics()
9313:             for phase_id, instr in self._phase_instrumentation.items()
9314:         }
9315:
9316:     def export_metrics(self) -> dict[str, Any]:
9317:         """Export metrics."""
9318:         abort_timestamp = self.abort_signal.get_timestamp()
9319:
9320:         return {
9321:             "timestamp": datetime.utcnow().isoformat(),
9322:             "phase_metrics": self.get_phase_metrics(),
9323:             "resource_usage": self.resource_limits.get_usage_history(),
9324:             "abort_status": {
9325:                 "is_aborted": self.abort_signal.is_aborted(),
9326:                 "reason": self.abort_signal.get_reason(),
9327:                 "timestamp": abort_timestamp.isoformat() if abort_timestamp else None,
9328:             },
9329:             "phase_status": dict(self._phase_status),
9330:         }
9331:
9332:     # ========================================================================
9333:     # PHASE IMPLEMENTATIONS
9334:     # ========================================================================
9335:
9336:     def _load_configuration(self) -> dict[str, Any]:
9337:         """FASE 0: Load configuration."""
9338:         self._ensure_not_aborted()
9339:         instrumentation = self._phase_instrumentation[0]
9340:         start = time.perf_counter()
9341:
9342:         monolith = _normalize_monolith_for_hash(self._monolith_data)
9343:         monolith_hash = hashlib.sha256(
9344:             json.dumps(monolith, sort_keys=True, ensure_ascii=False, separators=(",", ":"))
9345:             .encode("utf-8")
9346:         ).hexdigest()
9347:
9348:         micro_questions = monolith["blocks"].get("micro_questions", [])
9349:         meso_questions = monolith["blocks"].get("meso_questions", [])
9350:         macro_question = monolith["blocks"].get("macro_question", {})
9351:
9352:         question_total = len(micro_questions) + len(meso_questions) + (1 if macro_question else 0)
```

```
9353:
9354:         if question_total != EXPECTED_QUESTION_COUNT:
9355:             logger.warning(f"Question count: expected {EXPECTED_QUESTION_COUNT}, got {question_total}")
9356:             instrumentation.record_warning("integrity", f"Question count: {question_total}")
9357:
9358:         aggregation_settings = AggregationSettings.from_monolith(monolith)
9359:
9360:         duration = time.perf_counter() - start
9361:         instrumentation.increment(latency=duration)
9362:
9363:         return {
9364:             "monolith": monolith,
9365:             "monolith_sha256": monolith_hash,
9366:             "micro_questions": micro_questions,
9367:             "meso_questions": meso_questions,
9368:             "macro_question": macro_question,
9369:             "_aggregation_settings": aggregation_settings,
9370:         }
9371:
9372:     def _ingest_document(self, pdf_path: str, config: dict[str, Any]) -> PreprocessedDocument:
9373:         """FASE 1: Ingest document using SPC pipeline."""
9374:         self._ensure_not_aborted()
9375:         instrumentation = self._phase_instrumentation[1]
9376:         start = time.perf_counter()
9377:
9378:         document_id = os.path.splitext(os.path.basename(pdf_path))[0] or "doc_1"
9379:
9380:         try:
9381:             from farfan_pipeline.processing.spc_ingestion import CPPIngestionPipeline
9382:
9383:             pipeline = CPPIngestionPipeline()
9384:             canon_package = asyncio.run(pipeline.process(
9385:                 document_path=Path(pdf_path),
9386:                 document_id=document_id
9387:             ))
9388:
9389:             preprocessed = PreprocessedDocument.ensure(
9390:                 canon_package,
9391:                 document_id=document_id,
9392:                 use_spc_ingestion=True
9393:             )
9394:
9395:             if not preprocessed.raw_text or not preprocessed.raw_text.strip():
9396:                 raise ValueError("Empty document after ingestion")
9397:
9398:             actual_chunk_count = preprocessed.metadata.get("chunk_count", 0)
9399:             if actual_chunk_count != P01_EXPECTED_CHUNK_COUNT:
9400:                 raise ValueError(
9401:                     f"P01 validation failed: expected {P01_EXPECTED_CHUNK_COUNT} chunks, "
9402:                     f"got {actual_chunk_count}"
9403:                 )
9404:
9405:             for i, chunk in enumerate(preprocessed.chunks):
9406:                 if not getattr(chunk, "policy_area_id", None):
9407:                     raise ValueError(f"Chunk {i} missing policy_area_id")
9408:                 if not getattr(chunk, "dimension_id", None):
```

```
                    raise ValueError(f"Chunk {i} missing dimension_id")

            logger.info(f"â\234\223 P01-ES v1.0 validation passed: {actual_chunk_count} chunks")

        except Exception as e:
            instrumentation.record_error("ingestion", str(e))
            raise RuntimeError(f"Document ingestion failed: {e}") from e

        duration = time.perf_counter() - start
        instrumentation.increment(latency=duration)

        return preprocessed

    async def _execute_micro_questions_async(
        self, document: PreprocessedDocument, config: dict[str, Any]
    ) -> list[MicroQuestionRun]:
        """FASE 2: Execute micro-questions (STUB - requires your implementation)."""
        self._ensure_not_aborted()
        instrumentation = self._phase_instrumentation[2]

        micro_questions = config.get("micro_questions", [])
        instrumentation.start(items_total=len(micro_questions))

        logger.warning("Phase 2 stub - add your executor logic here")

        results: list[MicroQuestionRun] = []
        return results

    async def _score_micro_results_async(
        self, micro_results: list[MicroQuestionRun], config: dict[str, Any]
    ) -> list[ScoredMicroQuestion]:
        """FASE 3: Score results (STUB - requires your implementation)."""
        self._ensure_not_aborted()
        instrumentation = self._phase_instrumentation[3]

        instrumentation.start(items_total=len(micro_results))

        logger.warning("Phase 3 stub - add your scoring logic here")

        scored_results: list[ScoredMicroQuestion] = []
        return scored_results

    async def _aggregate_dimensions_async(
        self, scored_results: list[ScoredMicroQuestion], config: dict[str, Any]
    ) -> list[DimensionScore]:
        """FASE 4: Aggregate dimensions (STUB - requires your implementation)."""
        self._ensure_not_aborted()
        instrumentation = self._phase_instrumentation[4]

        instrumentation.start(items_total=6)

        logger.warning("Phase 4 stub - add your aggregation logic here")

        dimension_scores: list[DimensionScore] = []
        return dimension_scores

```

```
9465:     async def _aggregate_policy_areas_async(
9466:         self, dimension_scores: list[DimensionScore], config: dict[str, Any]
9467:     ) -> list[AreaScore]:
9468:         """FASE 5: Aggregate policy areas (STUB - requires your implementation)."""
9469:         self._ensure_not_aborted()
9470:         instrumentation = self._phase_instrumentation[5]
9471:
9472:         instrumentation.start(items_total=10)
9473:
9474:         logger.warning("Phase 5 stub - add your aggregation logic here")
9475:
9476:         area_scores: list[AreaScore] = []
9477:         return area_scores
9478:
9479:     def _aggregate_clusters(
9480:         self, policy_area_scores: list[AreaScore], config: dict[str, Any]
9481:     ) -> list[ClusterScore]:
9482:         """FASE 6: Aggregate clusters (STUB - requires your implementation)."""
9483:         self._ensure_not_aborted()
9484:         instrumentation = self._phase_instrumentation[6]
9485:
9486:         instrumentation.start(items_total=4)
9487:
9488:         logger.warning("Phase 6 stub - add your aggregation logic here")
9489:
9490:         cluster_scores: list[ClusterScore] = []
9491:         return cluster_scores
9492:
9493:     def _evaluate_macro(
9494:         self, cluster_scores: list[ClusterScore], config: dict[str, Any]
9495:     ) -> MacroEvaluation:
9496:         """FASE 7: Evaluate macro (STUB - requires your implementation)."""
9497:         self._ensure_not_aborted()
9498:         instrumentation = self._phase_instrumentation[7]
9499:
9500:         instrumentation.start(items_total=1)
9501:
9502:         logger.warning("Phase 7 stub - add your macro logic here")
9503:
9504:         macro_eval = MacroEvaluation(
9505:             macro_score=0.0,
9506:             macro_score_normalized=0.0,
9507:             clusters=[]
9508:         )
9509:         return macro_eval
9510:
9511:     async def _generate_recommendations(
9512:         self, macro_result: MacroEvaluation, config: dict[str, Any]
9513:     ) -> dict[str, Any]:
9514:         """FASE 8: Generate recommendations (STUB - requires your implementation)."""
9515:         self._ensure_not_aborted()
9516:         instrumentation = self._phase_instrumentation[8]
9517:
9518:         instrumentation.start(items_total=1)
9519:
9520:         logger.warning("Phase 8 stub - add your recommendation logic here")
```

```
9521:
9522:         recommendations = {
9523:             "status": "stub",
9524:             "macro_score": macro_result.macro_score,
9525:         }
9526:         return recommendations
9527:
9528:     def _assemble_report(
9529:         self, recommendations: dict[str, Any], config: dict[str, Any]
9530:     ) -> dict[str, Any]:
9531:         """FASE 9: Assemble report (STUB - requires your implementation)."""
9532:         self._ensure_not_aborted()
9533:         instrumentation = self._phase_instrumentation[9]
9534:
9535:         instrumentation.start(items_total=1)
9536:
9537:         logger.warning("Phase 9 stub - add your report logic here")
9538:
9539:         report = {
9540:             "status": "stub",
9541:             "recommendations": recommendations,
9542:         }
9543:         return report
9544:
9545:     async def _format_and_export(
9546:         self, report: dict[str, Any], config: dict[str, Any]
9547:     ) -> dict[str, Any]:
9548:         """FASE 10: Format and export (STUB - requires your implementation)."""
9549:         self._ensure_not_aborted()
9550:         instrumentation = self._phase_instrumentation[10]
9551:
9552:         instrumentation.start(items_total=1)
9553:
9554:         logger.warning("Phase 10 stub - add your export logic here")
9555:
9556:         export_payload = {
9557:             "status": "stub",
9558:             "report": report,
9559:         }
9560:         return export_payload
9561:
9562:
9563: # ============================================================================
9564: # EXPORTS
9565: # ============================================================================
9566:
9567: __all__ = [
9568:     "Orchestrator",
9569:     "MethodExecutor",
9570:     "AbortSignal",
9571:     "AbortRequested",
9572:     "ResourceLimits",
9573:     "PhaseInstrumentation",
9574:     "PhaseResult",
9575:     "MicroQuestionRun",
9576:     "ScoredMicroQuestion",
```

```
9577:     "Evidence",
9578:     "MacroEvaluation",
9579: ]
9580:
9581:
```