```
scripts/normalize_contract_patterns.py

#!/usr/bin/env python3
"""Normalize v3 executor contract patterns.

What this does (batch-safe, deterministic):
- For every Q###.v3.json in Phase_two executor_contracts/specialized/
  - Ensure each question_context.patterns[] has a usable "pattern" string.
          - If a pattern has pattern_ref but no pattern, resolve it from
canonic_questionnaire_central/pattern_registry.json
  - Fill missing match_type from registry when possible
  - Preserve all existing fields (no lossy transform)
     - Recompute identity.contract_hash using canonical sha256 rule used by
ContractUpdateValidator
  - Update identity.updated_at

This fixes the core issue where contracts carried pattern_ref without the actual regex,
which makes
"patterns" effectively non-executable at runtime.

Run:
  python3 scripts/normalize_contract_patterns.py

Optional:
  DRY_RUN=1 python3 scripts/normalize_contract_patterns.py
"""

from __future__ import annotations

import hashlib
import json
import os
from datetime import datetime, timezone
from pathlib import Path
from typing import Any


REPO_ROOT = Path(__file__).resolve().parents[1]
CONTRACTS_DIR = (
    REPO_ROOT
    / "src"
    / "canonic_phases"
    / "Phase_two"
    / "json_files_phase_two"
    / "executor_contracts"
    / "specialized"
)
PATTERN_REGISTRY_PATH     =     REPO_ROOT     /     "canonic_questionnaire_central"     /
"pattern_registry.json"


def _utc_now_iso() -> str:
    return datetime.now(timezone.utc).isoformat()
```

```
def _load_pattern_registry(path: Path) -> dict[str, dict[str, Any]]:
    raw = json.loads(path.read_text(encoding="utf-8"))
    if not isinstance(raw, list):
                    raise   TypeError(f"pattern_registry.json   must   be   a   list,   got
{type(raw).__name__}")
    idx: dict[str, dict[str, Any]] = {}
    for item in raw:
        if not isinstance(item, dict):
            continue
        pid = item.get("pattern_id") or item.get("id")
        if isinstance(pid, str) and pid:
            idx[pid] = item
    return idx


def _compute_contract_hash(contract: dict[str, Any]) -> str:
    # Mirror ContractUpdateValidator behavior to avoid hash drift.
    temp = json.loads(json.dumps(contract))
    try:
        del temp["identity"]["contract_hash"]
    except Exception:
        pass
    contract_str = json.dumps(temp, sort_keys=True)
    return hashlib.sha256(contract_str.encode()).hexdigest()


def _normalize_patterns_in_contract(
    contract: dict[str, Any],
    registry: dict[str, dict[str, Any]],
) -> tuple[dict[str, Any], list[str]]:
    warnings: list[str] = []

    qc = contract.get("question_context")
    if not isinstance(qc, dict):
        return contract, ["missing_or_invalid_question_context"]

    pats = qc.get("patterns")
    if not isinstance(pats, list):
        return contract, ["missing_or_invalid_question_context.patterns"]

    for idx, pat in enumerate(pats):
        if not isinstance(pat, dict):
            warnings.append(f"patterns[{idx}] not an object")
            continue

        pattern_ref = pat.get("pattern_ref")
        pattern_str = pat.get("pattern")

            # Resolve missing pattern from global registry when pattern_ref points to
PAT-xxxx
                if  (not  isinstance(pattern_str,  str)  or  not  pattern_str.strip())  and
isinstance(pattern_ref, str) and pattern_ref:
            src = registry.get(pattern_ref)
```

```
            if src and isinstance(src.get("pattern"), str):
                pat["pattern"] = src["pattern"]
                # Only fill match_type if missing/null
                                if  not  isinstance(pat.get("match_type"),  str)  or  not
pat.get("match_type"):
                    if isinstance(src.get("match_type"), str):
                        pat["match_type"] = src["match_type"]
            else:
                warnings.append(f"patterns[{idx}] unresolved pattern_ref={pattern_ref}")

        # If match_type still missing, default to REGEX (schema-friendly)
        if not isinstance(pat.get("match_type"), str) or not pat.get("match_type"):
            pat["match_type"] = "REGEX"

    qc["patterns"] = pats
    contract["question_context"] = qc

    ident = contract.get("identity")
    if isinstance(ident, dict):
        ident["updated_at"] = _utc_now_iso()
        contract["identity"] = ident

    # Recompute hash after normalization.
    if isinstance(contract.get("identity"), dict):
        contract["identity"]["contract_hash"] = _compute_contract_hash(contract)

    return contract, warnings


def main() -> int:
    if not CONTRACTS_DIR.exists():
        raise FileNotFoundError(str(CONTRACTS_DIR))
    if not PATTERN_REGISTRY_PATH.exists():
        raise FileNotFoundError(str(PATTERN_REGISTRY_PATH))

    dry_run = os.getenv("DRY_RUN", "0").strip() in {"1", "true", "TRUE", "yes", "YES"}

    registry = _load_pattern_registry(PATTERN_REGISTRY_PATH)

    contract_files = sorted(CONTRACTS_DIR.glob("Q*.v3.json"))
    if not contract_files:
        raise RuntimeError(f"No v3 contracts found in {CONTRACTS_DIR}")

    total = 0
    changed = 0
    total_warnings: list[str] = []

    for path in contract_files:
        total += 1
        original_text = path.read_text(encoding="utf-8")
        contract = json.loads(original_text)
        if not isinstance(contract, dict):
            total_warnings.append(f"{path.name}: not a JSON object")
            continue
```

```
        normalized, warnings = _normalize_patterns_in_contract(contract, registry)
        total_warnings.extend(f"{path.name}: {w}" for w in warnings)

         new_text = json.dumps(normalized, ensure_ascii=False, indent=2, sort_keys=False)
+ "\n"

        if new_text != original_text:
            changed += 1
            if not dry_run:
                path.write_text(new_text, encoding="utf-8")

    print(f"contracts_total={total} changed={changed} dry_run={dry_run}")
    if total_warnings:
        # Print only the first chunk to keep output readable
        print(f"warnings_count={len(total_warnings)}")
        for w in total_warnings[:50]:
            print(f"WARN: {w}")
        if len(total_warnings) > 50:
            print("... (more warnings truncated)")

    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

scripts/normalize_contracts_for_nexus.py

```
#!/usr/bin/env python3
"""Normalize v3 executor contracts for EvidenceNexus (definitive alignment).

Goals (batch-safe, deterministic):
- Remove/neutralize legacy EvidenceAssembler/EvidenceValidator references in contracts.
- Update wiring metadata to refer to EvidenceNexus + ValidationEngine.
- Enrich contract patterns with explicit policy_area for scope coherence.
- Make expected_elements usable as validation gates by ensuring minimum defaults.
- Copy failure_contract into question_context when only present in error_handling.
- Recompute identity.contract_hash (sha256 over canonical JSON, excluding old hash) and
update identity.updated_at.

Run:
  python3 scripts/normalize_contracts_for_nexus.py

Optional:
  DRY_RUN=1 python3 scripts/normalize_contracts_for_nexus.py
"""

from __future__ import annotations

import hashlib
import json
import os
from datetime import datetime, timezone
from pathlib import Path
from typing import Any


REPO_ROOT = Path(__file__).resolve().parents[1]
CONTRACTS_DIR = (
    REPO_ROOT
    / "src"
    / "canonic_phases"
    / "Phase_two"
    / "json_files_phase_two"
    / "executor_contracts"
    / "specialized"
)


def _utc_now_iso() -> str:
    return datetime.now(timezone.utc).isoformat()


def _compute_contract_hash(contract: dict[str, Any]) -> str:
    # Mirror ContractUpdateValidator behavior to avoid hash drift.
    temp = json.loads(json.dumps(contract))
    try:
        del temp["identity"]["contract_hash"]
    except Exception:
        pass
```

```
    contract_str = json.dumps(temp, sort_keys=True)
    return hashlib.sha256(contract_str.encode()).hexdigest()


_REPLACEMENTS: tuple[tuple[str, str], ...] = (
                                      ("farfan_core.core.orchestrator.evidence_assembler",
"canonic_phases.Phase_two.evidence_nexus"),
                                      ("farfan_core.core.orchestrator.evidence_validator",
"canonic_phases.Phase_two.evidence_nexus"),
                                       ("farfan_core.core.orchestrator.evidence_registry",
"canonic_phases.Phase_two.evidence_nexus"),
    ("EvidenceAssembler", "EvidenceNexus"),
    ("EvidenceValidator", "ValidationEngine"),
)


def _rewrite_strings(obj: Any) -> Any:
    """Recursively rewrite legacy strings inside an arbitrary JSON structure."""
    if isinstance(obj, str):
        out = obj
        for old, new in _REPLACEMENTS:
            out = out.replace(old, new)
        return out
    if isinstance(obj, list):
        return [_rewrite_strings(x) for x in obj]
    if isinstance(obj, dict):
        return {k: _rewrite_strings(v) for k, v in obj.items()}
    return obj


def _normalize_contract(contract: dict[str, Any]) -> tuple[dict[str, Any], list[str]]:
    warnings: list[str] = []

    identity = contract.get("identity")
    if not isinstance(identity, dict):
        return contract, ["missing_or_invalid_identity"]

    policy_area_id = identity.get("policy_area_id")
    if not isinstance(policy_area_id, str) or not policy_area_id:
        warnings.append("missing_identity.policy_area_id")
        policy_area_id = ""

     # 1) Rewrite legacy strings everywhere (docs, traceability, human_answer_structure,
etc.)
    contract = _rewrite_strings(contract)

    # 2) Normalize evidence_assembly wiring metadata to Nexus (non-breaking: Nexus reads
assembly_rules)
    evidence_assembly = contract.get("evidence_assembly")
    if isinstance(evidence_assembly, dict):
        evidence_assembly.setdefault("engine", "EVIDENCE_NEXUS")
        evidence_assembly["module"] = "canonic_phases.Phase_two.evidence_nexus"
        evidence_assembly["class_name"] = "EvidenceNexus"
          # There is no direct EvidenceNexus.assemble; process/process_evidence is the
```

```
canonical entry point.
            evidence_assembly["method_name"] = evidence_assembly.get("method_name") or
"process"
        contract["evidence_assembly"] = evidence_assembly

    # 3) Normalize validation_rules wiring metadata to ValidationEngine (non-breaking)
    validation_rules = contract.get("validation_rules")
    if isinstance(validation_rules, dict):
        validation_rules.setdefault("engine", "VALIDATION_ENGINE")
        validation_rules["module"] = "canonic_phases.Phase_two.evidence_nexus"
        validation_rules["class_name"] = "ValidationEngine"
              validation_rules["method_name"] = validation_rules.get("method_name") or
"validate"
        contract["validation_rules"] = validation_rules

        # 4) Ensure question_context has policy-scoped patterns and minimums for
expected_elements
    qc = contract.get("question_context")
    if not isinstance(qc, dict):
        warnings.append("missing_or_invalid_question_context")
        qc = {}

    pats = qc.get("patterns")
    if isinstance(pats, list):
        for p in pats:
            if isinstance(p, dict):
                    # Scope coherence: question-level patterns belong to the contract's
policy area.
                p.setdefault("policy_area", policy_area_id)
                # Ensure match_type exists (schema-friendly)
                if not isinstance(p.get("match_type"), str) or not p.get("match_type"):
                    p["match_type"] = "REGEX"
        qc["patterns"] = pats
    else:
        warnings.append("missing_or_invalid_question_context.patterns")

    elems = qc.get("expected_elements")
    if isinstance(elems, list):
        for e in elems:
            if not isinstance(e, dict):
                continue
            required = bool(e.get("required", False))
            # Make "required" measurable in a deterministic way.
            if required and "minimum" not in e:
                e["minimum"] = 1
            # Some monolith entries omit required; keep as-is
        qc["expected_elements"] = elems
    else:
        warnings.append("missing_or_invalid_question_context.expected_elements")

    # Ensure validations exists (monolith compatible)
    if "validations" not in qc or not isinstance(qc.get("validations"), dict):
        qc["validations"] = {}
```

```
    # 5) Copy failure_contract into question_context if only present in error_handling
    error_handling = contract.get("error_handling")
    if isinstance(error_handling, dict):
        fc = error_handling.get("failure_contract")
        if isinstance(fc, dict) and "failure_contract" not in qc:
            qc["failure_contract"] = fc

    contract["question_context"] = qc

    # 6) Update timestamps and contract hash
    identity["updated_at"] = _utc_now_iso()
    contract["identity"] = identity
    contract["identity"]["contract_hash"] = _compute_contract_hash(contract)

    return contract, warnings


def main() -> int:
    if not CONTRACTS_DIR.exists():
        raise FileNotFoundError(str(CONTRACTS_DIR))

    dry_run = os.getenv("DRY_RUN", "0").strip() in {"1", "true", "TRUE", "yes", "YES"}

    contract_files = sorted(CONTRACTS_DIR.glob("Q*.v3.json"))
    if not contract_files:
        raise RuntimeError(f"No v3 contracts found in {CONTRACTS_DIR}")

    total = 0
    changed = 0
    total_warnings: list[str] = []

    for path in contract_files:
        total += 1
        original_text = path.read_text(encoding="utf-8")
        contract = json.loads(original_text)
        if not isinstance(contract, dict):
            total_warnings.append(f"{path.name}: not a JSON object")
            continue

        normalized, warnings = _normalize_contract(contract)
        total_warnings.extend(f"{path.name}: {w}" for w in warnings)

        new_text = json.dumps(normalized, ensure_ascii=False, indent=2, sort_keys=False)
+ "\n"
        if new_text != original_text:
            changed += 1
            if not dry_run:
                path.write_text(new_text, encoding="utf-8")

    print(f"contracts_total={total} changed={changed} dry_run={dry_run}")
    if total_warnings:
        print(f"warnings_count={len(total_warnings)}")
        for w in total_warnings[:50]:
            print(f"WARN: {w}")
```

```
        if len(total_warnings) > 50:
            print("... (more warnings truncated)")

    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

scripts/populate_signal_requirements.py

```
#!/usr/bin/env python3
"""
Populate signal_requirements for V3 executor contracts based on policy areas.

This script updates the signal_requirements section of each contract to include
appropriate mandatory and optional signals based on the policy area and dimension.

Usage:
    python scripts/populate_signal_requirements.py [--dry-run]
"""

import argparse
import json
import sys
from pathlib import Path


# Policy area-specific signal mappings
POLICY_AREA_SIGNALS = {
    "PA01": {  # Women's Rights & Gender Equality
        "mandatory": ["gender_baseline_data", "vbg_statistics", "policy_coverage"],
        "optional": ["temporal_series", "source_validation", "territorial_scope"]
    },
    "PA02": {  # Rural Development
        "mandatory": ["rural_indicators", "land_tenure_data", "agricultural_policy"],
        "optional": ["infrastructure_gaps", "market_access", "subsidy_programs"]
    },
    "PA03": {  # Education
        "mandatory": ["enrollment_rates", "quality_indicators", "coverage_data"],
        "optional": ["infrastructure_status", "teacher_ratios", "dropout_rates"]
    },
    "PA04": {  # Health
        "mandatory": ["health_coverage", "mortality_rates", "service_availability"],
        "optional": ["disease_prevalence", "vaccination_rates", "infrastructure"]
    },
    "PA05": {  # Infrastructure
        "mandatory": ["infrastructure_inventory", "coverage_gaps", "investment_plans"],
        "optional": ["maintenance_status", "connectivity", "service_quality"]
    },
    "PA06": {  # Economic Development
                            "mandatory":    ["economic_indicators",    "employment_data",
"sectoral_distribution"],
        "optional": ["investment_flows", "productivity_metrics", "gdp_municipal"]
    },
    "PA07": {  # Environment
        "mandatory": ["environmental_baseline", "protection_areas", "risk_zones"],
        "optional": ["deforestation_rates", "water_quality", "biodiversity"]
    },
    "PA08": {  # Governance
                    "mandatory":  ["institutional_capacity",  "participation_mechanisms",
"transparency"],
                        "optional":   ["corruption_indicators",   "citizen_satisfaction",
```

```
"planning_quality"]
    },
    "PA09": {  # Security & Justice
        "mandatory": ["crime_statistics", "justice_access", "security_coverage"],
                        "optional":   ["conflict_indicators",   "institutional_presence",
"victimization_rates"]
    },
    "PA10": {  # Culture & Tourism
                         "mandatory":   ["cultural_assets",   "tourism_infrastructure",
"heritage_protection"],
        "optional": ["visitor_statistics", "cultural_programming", "economic_impact"]
    }
}

# Dimension-specific signals (apply across all policy areas)
DIMENSION_SIGNALS = {
    "DIM01": {  # Diagnostic Quality
        "mandatory": ["baseline_completeness", "data_sources"],
        "optional": ["temporal_coverage", "geographic_scope"]
    },
    "DIM02": {  # Causal Logic
        "mandatory": ["causal_chains", "intervention_logic"],
        "optional": ["theory_of_change", "assumptions"]
    },
    "DIM03": {  # Product Planning
        "mandatory": ["product_targets", "budget_allocation"],
        "optional": ["implementation_schedule", "responsible_entities"]
    },
    "DIM04": {  # Outcome Definition
        "mandatory": ["outcome_indicators", "measurement_validity"],
        "optional": ["composite_metrics", "verification_sources"]
    },
    "DIM05": {  # Impact Ambition
        "mandatory": ["long_term_vision", "transformative_potential"],
        "optional": ["sustainability_mechanisms", "scalability"]
    },
    "DIM06": {  # Territorial Context
        "mandatory": ["territorial_diagnosis", "differential_needs"],
        "optional": ["participation_evidence", "equity_considerations"]
    }
}


def populate_signal_requirements(contract: dict, dry_run: bool = False) -> tuple[bool,
str]:
    """Populate signal_requirements for a contract.

    Args:
        contract: Contract dictionary
        dry_run: If True, don't modify contract

    Returns:
        (modified, message) tuple
    """
```

```
    # Extract policy area and dimension
    policy_area_id = contract.get("identity", {}).get("policy_area_id")
    dimension_id = contract.get("identity", {}).get("dimension_id")

    if not policy_area_id or not dimension_id:
        return False, "Missing policy_area_id or dimension_id"

    # Get signals for policy area and dimension
    pa_signals = POLICY_AREA_SIGNALS.get(policy_area_id, {})
    dim_signals = DIMENSION_SIGNALS.get(dimension_id, {})

    # Combine mandatory signals (union)
    mandatory_signals = list(set(
        pa_signals.get("mandatory", []) +
        dim_signals.get("mandatory", [])
    ))

    # Combine optional signals (union)
    optional_signals = list(set(
        pa_signals.get("optional", []) +
        dim_signals.get("optional", [])
    ))

    # Check if update needed
    current_sig_req = contract.get("signal_requirements", {})
    current_mandatory = current_sig_req.get("mandatory_signals", [])
    current_optional = current_sig_req.get("optional_signals", [])

    if (set(current_mandatory) == set(mandatory_signals) and
        set(current_optional) == set(optional_signals)):
        return False, "Signal requirements already correct"

    # Update signal_requirements
    if not dry_run:
        if "signal_requirements" not in contract:
            contract["signal_requirements"] = {}

        contract["signal_requirements"]["mandatory_signals"] = sorted(mandatory_signals)
        contract["signal_requirements"]["optional_signals"] = sorted(optional_signals)

        # Keep other fields
        if "signal_aggregation" not in contract["signal_requirements"]:
            contract["signal_requirements"]["signal_aggregation"] = "weighted_mean"
        if "minimum_signal_threshold" not in contract["signal_requirements"]:
            contract["signal_requirements"]["minimum_signal_threshold"] = 0.0

              return    True,    f"Updated    with    {len(mandatory_signals)}    mandatory,
{len(optional_signals)} optional signals"


def main():
    parser = argparse.ArgumentParser(
        description="Populate signal_requirements for V3 executor contracts"
    )
```

```
    parser.add_argument(
        "--dry-run",
        action="store_true",
        help="Show what would be updated without making changes"
    )
    parser.add_argument(
        "--contracts-dir",
        type=Path,
        help="Path to contracts directory (default: auto-detect)"
    )

    args = parser.parse_args()

    # Find contracts directory
    if args.contracts_dir:
        contracts_dir = args.contracts_dir
    else:
        script_dir = Path(__file__).parent
        project_root = script_dir.parent
            contracts_dir = project_root / "src" / "canonic_phases" / "Phase_two" /
"json_files_phase_two" / "executor_contracts" / "specialized"

    if not contracts_dir.exists():
        print(f"? Contracts directory not found: {contracts_dir}", file=sys.stderr)
        sys.exit(1)

    # Find all V3 contracts
    contract_files = sorted(contracts_dir.glob("Q*.v3.json"))

    if not contract_files:
        print(f"? No V3 contracts found in {contracts_dir}", file=sys.stderr)
        sys.exit(1)

    print(f"Found {len(contract_files)} V3 contracts")
    print()

    # Process contracts
    updated = 0
    already_correct = 0
    errors = 0

    for contract_path in contract_files:
        contract_id = contract_path.stem.replace(".v3", "")

        try:
            # Load contract
            with open(contract_path, "r", encoding="utf-8") as f:
                contract = json.load(f)

            # Populate signal requirements
                        modified, message = populate_signal_requirements(contract,
dry_run=args.dry_run)

            # Treat missing identifiers as an error, not "already correct"
```

```
                    if not modified and message.startswith("Missing policy_area_id or
dimension_id"):
                errors += 1
                print(f"? {contract_id}: {message}")
                continue

            if modified:
                updated += 1
                print(f"? {contract_id}: {message}")

                if not args.dry_run:
                    # Write back
                    with open(contract_path, "w", encoding="utf-8") as f:
                        json.dump(contract, f, indent=2, ensure_ascii=False)
                        f.write("\n")
            else:
                already_correct += 1
                # Only print every 50th to reduce output
                if already_correct % 50 == 1:
                    print(f"? {contract_id}: {message}")

        except Exception as e:
            errors += 1
            print(f"? {contract_id}: Error: {e}")

    print()
    print("=" * 60)
    print("Summary:")
    print(f"  Total contracts: {len(contract_files)}")
    print(f"  Updated: {updated}")
    print(f"  Already correct: {already_correct}")
    print(f"  Errors: {errors}")

    if args.dry_run:
        print()
        print("DRY RUN - No files were modified")

    print("=" * 60)

    sys.exit(0 if errors == 0 else 2)


if __name__ == "__main__":
    main()
```

scripts/rollback_contract.py

```
#!/usr/bin/env python3
"""
Contract Rollback Utility
Rollback contracts to previous backup versions.
"""

from __future__ import annotations

import argparse
import json
import sys
from datetime import datetime
from pathlib import Path

sys.path.insert(0, str(Path(__file__).parent.parent))

from scripts.contract_remediator import ContractBackupManager, ContractDiffGenerator


def list_backups(backup_dir: Path, contract_name: str | None = None) -> None:
    """List available backups."""
    manager = ContractBackupManager(backup_dir)

    if contract_name:
        backups = manager.list_backups(contract_name)
        if not backups:
            print(f"No backups found for {contract_name}")
            return

        print(f"\nBackups for {contract_name}:")
        for i, backup in enumerate(backups):
            size = backup.stat().st_size
            mtime = datetime.fromtimestamp(backup.stat().st_mtime)
            print(f"  [{i}] {backup.name}")
            print(f"      Size: {size:,} bytes")
            print(f"      Modified: {mtime.strftime('%Y-%m-%d %H:%M:%S')}")
    else:
        all_backups = sorted(backup_dir.glob("*_backup_*.json"))
        if not all_backups:
            print("No backups found")
            return

        print("\nAll backups:")
        contracts = {}
        for backup in all_backups:
            contract = backup.name.split("_backup_")[0]
            contracts.setdefault(contract, []).append(backup)

        for contract, backups in sorted(contracts.items()):
            print(f"\n{contract}: {len(backups)} backup(s)")
            for backup in backups[-3:]:
                mtime = datetime.fromtimestamp(backup.stat().st_mtime)
```

```
                print(f"  - {backup.name} ({mtime.strftime('%Y-%m-%d %H:%M:%S')})")


def show_backup_diff(backup_path: Path, current_path: Path) -> None:
    """Show diff between backup and current contract."""
    try:
        with open(backup_path) as f:
            backup_contract = json.load(f)

        with open(current_path) as f:
            current_contract = json.load(f)

        diff_gen = ContractDiffGenerator()
                    diff   =   diff_gen.generate_diff(backup_contract,   current_contract,
current_path.stem)

        if diff:
            print(f"\nDiff between backup and current version:")
            print(diff)
        else:
            print("\nNo differences found")

        changes = diff_gen.summarize_changes(backup_contract, current_contract)
        if any(changes.values()):
            print("\nSummary of changes:")
            if changes["fields_modified"]:
                print(f"  Modified: {', '.join(changes['fields_modified'][:5])}")
            if changes["fields_added"]:
                print(f"  Added: {', '.join(changes['fields_added'][:5])}")
            if changes["fields_removed"]:
                print(f"  Removed: {', '.join(changes['fields_removed'][:5])}")

    except Exception as e:
        print(f"Error generating diff: {e}")


def rollback_contract(
    backup_path: Path,
    target_path: Path,
    backup_dir: Path,
    dry_run: bool = False,
) -> None:
    """Rollback contract to backup version."""
    manager = ContractBackupManager(backup_dir)

    if not backup_path.exists():
        print(f"Error: Backup not found: {backup_path}")
        return

    if not target_path.exists():
        print(f"Warning: Target contract does not exist: {target_path}")

    if dry_run:
        print("\n? DRY RUN MODE - No files will be modified")
```

```
        show_backup_diff(backup_path, target_path)
        return

    print(f"\nCreating backup of current version...")
    current_backup = manager.backup_contract(target_path)
    print(f"  Backed up to: {current_backup.name}")

    print(f"\nRestoring from: {backup_path.name}")
    manager.restore_backup(backup_path, target_path)
    print(f"  ? Restored to: {target_path}")

    with open(target_path) as f:
        contract = json.load(f)
        identity = contract.get("identity", {})
        version = identity.get("contract_version", "unknown")
        question_id = identity.get("question_id", "unknown")

    print(f"\nRestored contract: {question_id} (version {version})")


def main():
    parser = argparse.ArgumentParser(
        description="Contract Rollback Utility",
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog="""
Examples:
  # List all backups
  python scripts/rollback_contract.py --list

  # List backups for specific contract
  python scripts/rollback_contract.py --list --contract Q002.v3

  # Show diff between backup and current
   python scripts/rollback_contract.py --diff --backup Q002_backup_20250101_120000.json
--contract Q002.v3.json

  # Rollback contract (with backup of current)
               python       scripts/rollback_contract.py       --rollback       --backup
Q002_backup_20250101_120000.json --contract Q002.v3.json

  # Dry run rollback
               python       scripts/rollback_contract.py       --rollback       --backup
Q002_backup_20250101_120000.json --contract Q002.v3.json --dry-run
        """,
    )

    parser.add_argument("--list", action="store_true", help="List available backups")
    parser.add_argument(
        "--diff", action="store_true", help="Show diff between backup and current"
    )
    parser.add_argument("--rollback", action="store_true", help="Rollback to backup")
    parser.add_argument(
              "--contract", type=str, help="Contract file name (e.g., Q002.v3.json or
Q002.v3)"
```

```
    )
    parser.add_argument("--backup", type=str, help="Backup file name")
    parser.add_argument("--dry-run", action="store_true", help="Preview rollback")
    parser.add_argument(
        "--backup-dir",
        type=Path,
        default=Path("backups/contracts"),
        help="Directory for contract backups",
    )
    parser.add_argument(
        "--contracts-dir",
        type=Path,
        default=Path(

"src/farfan_pipeline/phases/Phase_two/json_files_phase_two/executor_contracts/specialize
d"
        ),
        help="Directory containing contracts",
    )

    args = parser.parse_args()

    if not any([args.list, args.diff, args.rollback]):
        parser.error("Must specify --list, --diff, or --rollback")

    if args.list:
        contract_name = None
        if args.contract:
            contract_name = args.contract.replace(".json", "").replace(".v3", "")
        list_backups(args.backup_dir, contract_name)

    elif args.diff:
        if not args.backup or not args.contract:
            parser.error("--diff requires --backup and --contract")

        backup_path = args.backup_dir / args.backup
        contract_path = args.contracts_dir / args.contract
        if not contract_path.name.endswith(".json"):
            contract_path = args.contracts_dir / f"{args.contract}.json"

        show_backup_diff(backup_path, contract_path)

    elif args.rollback:
        if not args.backup or not args.contract:
            parser.error("--rollback requires --backup and --contract")

        backup_path = args.backup_dir / args.backup
        contract_path = args.contracts_dir / args.contract
        if not contract_path.name.endswith(".json"):
            contract_path = args.contracts_dir / f"{args.contract}.json"

        rollback_contract(backup_path, contract_path, args.backup_dir, args.dry_run)
```

```
if __name__ == "__main__":
    main()
```

scripts/run_policy_pipeline_verified.py

```
#!/usr/bin/env python3
"""
F.A.R.F.A.N Verified Pipeline Runner
====================================

Framework for Advanced Retrieval of Administrativa Narratives

Canonical entrypoint for executing the F.A.R.F.A.N policy analysis pipeline with
cryptographic verification and structured claim logging. This script is designed
to be machine-auditable and produces verifiable artifacts at every step.

Key Features:
- Computes SHA256 hashes of all inputs and outputs
- Emits structured JSON claims for all operations
- Generates verification_manifest.json with success status
- Enforces zero-trust validation principles
- No fabricated logs or unverifiable banners

Usage:
    python scripts/run_policy_pipeline_verified.py [--plan PLAN_PDF]

Requirements:
    - Input PDF must exist (default: data/plans/Plan_1.pdf)
    - All dependencies installed
    - Write access to artifacts/ directory
"""

import asyncio
import hashlib
import json
import os
import sys
import traceback
from dataclasses import asdict, dataclass
from datetime import datetime
from pathlib import Path
from typing import Any

# Ensure src/ is in Python path
REPO_ROOT = Path(__file__).parent.parent

# Import contract enforcement infrastructure
from saaaaaa.core.orchestrator.seed_registry import get_global_seed_registry
from saaaaaa.core.orchestrator.verification_manifest import (
    VerificationManifestBuilder,
    verify_manifest_integrity,
)
from src.orchestration.versions import get_all_versions


@dataclass
class ExecutionClaim:
```

```
    """Structured claim about a pipeline operation."""
    timestamp: str
    claim_type: str  # "start", "complete", "error", "artifact", "hash"
    component: str
    message: str
    data: dict[str, Any] | None = None

    def to_dict(self) -> dict[str, Any]:
        """Convert to dictionary for JSON serialization."""
        return asdict(self)


@dataclass
class VerificationManifest:
    """Complete verification manifest for pipeline execution."""
    success: bool
    execution_id: str
    start_time: str
    end_time: str
    input_pdf_path: str
    input_pdf_sha256: str
    artifacts_generated: list[str]
    artifact_hashes: dict[str, str]
    phases_completed: int
    phases_failed: int
    total_claims: int
    errors: list[str]

    def to_dict(self) -> dict[str, Any]:
        """Convert to dictionary for JSON serialization."""
        return asdict(self)


class VerifiedPipelineRunner:
    """Executes pipeline with cryptographic verification and claim logging."""

      def __init__(self, plan_pdf_path: Path, artifacts_dir: Path, questionnaire_path:
Path | None = None):
        """
        Initialize verified runner.

        Args:
            plan_pdf_path: Path to input PDF
            artifacts_dir: Directory for output artifacts
            questionnaire_path: Optional path to questionnaire file.
                                                If None, uses canonical path from
saaaaaa.config.paths.QUESTIONNAIRE_FILE
        """
        self.plan_pdf_path = plan_pdf_path
        self.artifacts_dir = artifacts_dir
        self.claims: list[ExecutionClaim] = []
        self.execution_id = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
        self.start_time = datetime.utcnow().isoformat()
        self.phases_completed = 0
```

```
    self.phases_failed = 0
    self.errors: list[str] = []
    self.orchestrator: Any = None  # Store orchestrator instance for metrics export

    # Set questionnaire path (explicit input, SIN_CARRETA compliance)
    if questionnaire_path is None:
        # Import here to avoid circular imports
        sys.path.insert(0, str(REPO_ROOT / 'src'))
        from saaaaaa.config.paths import QUESTIONNAIRE_FILE
        questionnaire_path = QUESTIONNAIRE_FILE

    self.questionnaire_path = questionnaire_path

    # Initialize seed registry for deterministic execution
    self.seed_registry = get_global_seed_registry()
    self.seed_registry = get_global_seed_registry()
    # Safely set identifiers regardless of SeedRegistry API shape
    if hasattr(self.seed_registry, "set_policy_unit_id"):
        self.seed_registry.set_policy_unit_id(f"plan1_{self.execution_id}")
    else:
        self.seed_registry.policy_unit_id = f"plan1_{self.execution_id}"
    if hasattr(self.seed_registry, "set_correlation_id"):
        self.seed_registry.set_correlation_id(self.execution_id)
    else:
        self.seed_registry.correlation_id = self.execution_id
    self.seed_registry.set_correlation_id(self.execution_id)

    # Initialize verification manifest builder
    self.manifest_builder = VerificationManifestBuilder()
    self.manifest_builder.set_versions(get_all_versions())

    # Ensure artifacts directory exists
    self.artifacts_dir.mkdir(parents=True, exist_ok=True)

def log_claim(self, claim_type: str, component: str, message: str,
              data: dict[str, Any] | None = None) -> None:
    """
    Log a structured claim.

    Args:
        claim_type: Type of claim (start, complete, error, artifact, hash)
        component: Component making the claim
        message: Human-readable message
        data: Optional structured data
    """
    claim = ExecutionClaim(
        timestamp=datetime.utcnow().isoformat(),
        claim_type=claim_type,
        component=component,
        message=message,
        data=data or {}
    )
    self.claims.append(claim)
```

```
        # Also print for real-time monitoring
        claim_json = json.dumps(claim.to_dict(), separators=(',', ':'))
        print(f"CLAIM: {claim_json}", flush=True)

    def compute_sha256(self, file_path: Path) -> str:
        """
        Compute SHA256 hash of a file.

        Args:
            file_path: Path to file

        Returns:
            Hex-encoded SHA256 hash
        """
        sha256_hash = hashlib.sha256()
        with open(file_path, "rb") as f:
            for byte_block in iter(lambda: f.read(4096), b""):
                sha256_hash.update(byte_block)
        return sha256_hash.hexdigest()

    def verify_input(self) -> bool:
        """
        Verify input PDF and questionnaire exist and compute hashes.

        Returns:
            True if all inputs are valid
        """
           self.log_claim("start", "input_verification", "Verifying input files (PDF +
questionnaire)")

        # Verify PDF
        if not self.plan_pdf_path.exists():
            error_msg = f"Input PDF not found: {self.plan_pdf_path}"
            self.log_claim("error", "input_verification", error_msg)
            self.errors.append(error_msg)
            return False

        # Verify questionnaire (CRITICAL for SIN_CARRETA compliance)
        if not self.questionnaire_path.exists():
            error_msg = f"Questionnaire file not found: {self.questionnaire_path}"
            self.log_claim("error", "input_verification", error_msg)
            self.errors.append(error_msg)
            return False

        # Compute PDF hash
        try:
            pdf_hash = self.compute_sha256(self.plan_pdf_path)
            self.input_pdf_sha256 = pdf_hash
            self.log_claim("hash", "input_verification",
                           f"Input PDF SHA256: {pdf_hash}",
                           {"file": str(self.plan_pdf_path), "hash": pdf_hash})
        except Exception as e:
            error_msg = f"Failed to hash input PDF: {e!s}"
            self.log_claim("error", "input_verification", error_msg)
```

```
            self.errors.append(error_msg)
            return False

        # Compute questionnaire hash (CRITICAL for determinism)
        try:
            questionnaire_hash = self.compute_sha256(self.questionnaire_path)
            self.questionnaire_sha256 = questionnaire_hash
            self.log_claim("hash", "input_verification",
                           f"Questionnaire SHA256: {questionnaire_hash}",
                                        {"file":  str(self.questionnaire_path),  "hash":
questionnaire_hash})
        except Exception as e:
            error_msg = f"Failed to hash questionnaire: {e!s}"
            self.log_claim("error", "input_verification", error_msg)
            self.errors.append(error_msg)
            return False

        self.log_claim("complete", "input_verification",
                       "Input verification successful (PDF + questionnaire)",
                       {"pdf_path": str(self.plan_pdf_path),
                        "questionnaire_path": str(self.questionnaire_path)})
        return True

    async def run_spc_ingestion(self) -> Any | None:
        """
        Run SPC (Smart Policy Chunks) ingestion phase - canonical phase-one.

        Passes explicit questionnaire_path to SPC pipeline for SIN_CARRETA compliance.

        Returns:
            SPC object if successful, None otherwise
        """
        self.log_claim("start", "spc_ingestion",
                       "Starting SPC ingestion (phase-one) with questionnaire",
                       {"questionnaire_path": str(self.questionnaire_path)})

        try:
            from saaaaaa.processing.spc_ingestion import CPPIngestionPipeline

            # Pass questionnaire_path explicitly (SIN_CARRETA: no hidden inputs)
            pipeline = CPPIngestionPipeline(questionnaire_path=self.questionnaire_path)
            cpp = await pipeline.process(self.plan_pdf_path)

            self.phases_completed += 1
            self.log_claim("complete", "spc_ingestion",
                           "SPC ingestion (phase-one) completed successfully",
                           {"phases_completed": self.phases_completed,
                            "questionnaire_path": str(self.questionnaire_path)})
            return cpp

        except Exception as e:
            self.phases_failed += 1
            error_msg = f"SPC ingestion failed: {e!s}"
            self.log_claim("error", "spc_ingestion", error_msg,
```

```
                        {"traceback": traceback.format_exc()})
            self.errors.append(error_msg)
            return None

    async def run_cpp_adapter(self, cpp: Any) -> Any | None:
        """
        Run SPC adapter to convert to PreprocessedDocument.

        Args:
            cpp: CPP/SPC object from ingestion

        Returns:
            PreprocessedDocument if successful, None otherwise
        """
        self.log_claim("start", "spc_adapter", "Starting SPC adaptation")

        try:
            from saaaaaa.utils.spc_adapter import SPCAdapter

            adapter = SPCAdapter()
            # Use the correct method name from SPCAdapter API
            preprocessed = adapter.to_preprocessed_document(cpp)

            self.phases_completed += 1
            self.log_claim("complete", "spc_adapter",
                           "SPC adaptation completed successfully",
                           {"phases_completed": self.phases_completed})
            return preprocessed

        except Exception as e:
            self.phases_failed += 1
            error_msg = f"SPC adaptation failed: {e!s}"
            self.log_claim("error", "spc_adapter", error_msg,
                           {"traceback": traceback.format_exc()})
            self.errors.append(error_msg)
            return None

    async def run_orchestrator(self, preprocessed_doc: Any) -> dict[str, Any] | None:
        """
        Run orchestrator with all phases.

        Args:
            preprocessed_doc: PreprocessedDocument

        Returns:
            Results dictionary if successful, None otherwise
        """
        self.log_claim("start", "orchestrator", "Starting orchestrator execution")

        try:
            from saaaaaa.core.orchestrator import Orchestrator
            from saaaaaa.core.orchestrator.factory import build_processor

            processor = build_processor()
```

```
            orchestrator = Orchestrator(processor=processor)

            # Store orchestrator instance for metrics export
            self.orchestrator = orchestrator

            # Run all phases
            results = await orchestrator.process(preprocessed_doc)

            # Count actual phases completed based on results
            if results and hasattr(results, '__dict__'):
                phase_attrs = [attr for attr in dir(results)
                               if not attr.startswith('_') and attr.endswith('_result')]
                completed_phases = sum(1 for attr in phase_attrs
                                       if getattr(results, attr, None) is not None)
                self.phases_completed += completed_phases
            else:
                # Fallback if we can't inspect results
                self.phases_completed += 1

            self.log_claim("complete", "orchestrator",
                           "Orchestrator execution completed successfully",
                           {"phases_completed": self.phases_completed})
            return results

        except Exception as e:
            self.phases_failed += 1
            error_msg = f"Orchestrator execution failed: {e!s}"
            self.log_claim("error", "orchestrator", error_msg,
                           {"traceback": traceback.format_exc()})
            self.errors.append(error_msg)
            return None

    def persist_orchestrator_metrics(self) -> tuple[list[str], dict[str, str]]:
        """
        Persist orchestrator metrics to artifacts directory.

        Exports and persists:
        - phase_metrics.json: Full PhaseInstrumentation.build_metrics() for each phase
        - resource_usage.jsonl: Serialized ResourceLimits.get_usage_history() snapshots
        - latency_histograms.json: Per-phase latency percentiles

        Returns:
            Tuple of (list of artifact paths, dict of artifact hashes)
        """
               self.log_claim("start",  "metrics_persistence",  "Persisting  orchestrator
metrics")

        artifacts = []
        artifact_hashes = {}

        if self.orchestrator is None:
            self.log_claim("error", "metrics_persistence",
                           "Orchestrator not available for metrics export")
            return artifacts, artifact_hashes
```

```
    try:
        # Import metrics persistence module
        sys.path.insert(0, str(REPO_ROOT / 'src'))
        from farfan_pipeline.orchestration.metrics_persistence import (
            persist_all_metrics,
            validate_metrics_schema,
        )

        # Export metrics from orchestrator
        metrics = self.orchestrator.export_metrics()

        # Validate metrics schema
        validation_errors = validate_metrics_schema(metrics)
        if validation_errors:
            error_msg = f"Metrics validation failed: {validation_errors}"
            self.log_claim("error", "metrics_persistence", error_msg,
                           {"validation_errors": validation_errors})
            self.errors.append(error_msg)
            return artifacts, artifact_hashes

        # Persist all metrics to artifacts directory
        written_files = persist_all_metrics(metrics, self.artifacts_dir)

        # Add written files to artifacts list and compute hashes
        for metric_type, file_path in written_files.items():
            artifacts.append(str(file_path))
            artifact_hashes[str(file_path)] = self.compute_sha256(file_path)
            self.log_claim("artifact", "metrics_persistence",
                           f"Persisted {metric_type} metrics",
                           {"file": str(file_path), "metric_type": metric_type})

        self.log_claim("complete", "metrics_persistence",
                       f"Successfully persisted {len(written_files)} metrics files",
                       {"files": list(written_files.keys())})

        return artifacts, artifact_hashes

    except Exception as e:
        error_msg = f"Failed to persist metrics: {e!s}"
        self.log_claim("error", "metrics_persistence", error_msg,
                       {"traceback": traceback.format_exc()})
        self.errors.append(error_msg)
        return artifacts, artifact_hashes

def save_artifacts(self, cpp: Any, preprocessed_doc: Any,
                   results: Any) -> tuple[list[str], dict[str, str]]:
    """
    Save artifacts and compute hashes.

    Args:
        cpp: CPP object
        preprocessed_doc: PreprocessedDocument
        results: Orchestrator results
```

```
        Returns:
            List of artifact file paths
        """
        self.log_claim("start", "artifact_generation", "Saving artifacts")

        artifacts = []
        artifact_hashes = {}

        try:
            # Save CPP metadata if available
            if cpp:
                cpp_metadata_path = self.artifacts_dir / "cpp_metadata.json"
                try:
                    with open(cpp_metadata_path, 'w') as f:
                        json.dump({
                            "execution_id": self.execution_id,
                            "cpp_generated": True,
                            "timestamp": datetime.utcnow().isoformat()
                        }, f, indent=2)
                    artifacts.append(str(cpp_metadata_path))
                                                artifact_hashes[str(cpp_metadata_path)]  =
self.compute_sha256(cpp_metadata_path)
                except Exception as e:
                    self.log_claim("error", "artifact_generation",
                                   f"Failed to save CPP metadata: {e!s}")

            # Save preprocessed document metadata
            if preprocessed_doc:
                                            doc_metadata_path   =   self.artifacts_dir   /
"preprocessed_doc_metadata.json"
                try:
                    with open(doc_metadata_path, 'w') as f:
                        json.dump({
                            "execution_id": self.execution_id,
                            "doc_generated": True,
                            "timestamp": datetime.utcnow().isoformat()
                        }, f, indent=2)
                    artifacts.append(str(doc_metadata_path))
                                                artifact_hashes[str(doc_metadata_path)]  =
self.compute_sha256(doc_metadata_path)
                except Exception as e:
                    self.log_claim("error", "artifact_generation",
                                   f"Failed to save doc metadata: {e!s}")

            # Save results summary
            if results:
                results_path = self.artifacts_dir / "results_summary.json"
                try:
                    with open(results_path, 'w') as f:
                        json.dump({
                            "execution_id": self.execution_id,
                            "results_generated": True,
                            "timestamp": datetime.utcnow().isoformat()
```

```
                    }, f, indent=2)
                artifacts.append(str(results_path))
                                                artifact_hashes[str(results_path)]   =
self.compute_sha256(results_path)
            except Exception as e:
                self.log_claim("error", "artifact_generation",
                               f"Failed to save results: {e!s}")

            # Save all claims
            claims_path = self.artifacts_dir / "execution_claims.json"
            with open(claims_path, 'w') as f:
                json.dump([claim.to_dict() for claim in self.claims], f, indent=2)
            artifacts.append(str(claims_path))
            artifact_hashes[str(claims_path)] = self.compute_sha256(claims_path)

            self.log_claim("complete", "artifact_generation",
                           f"Saved {len(artifacts)} artifacts",
                           {"artifact_count": len(artifacts)})

            return artifacts, artifact_hashes

        except Exception as e:
            error_msg = f"Failed to save artifacts: {e!s}"
            self.log_claim("error", "artifact_generation", error_msg)
            self.errors.append(error_msg)
            return artifacts, artifact_hashes

    def _calculate_chunk_metrics(self, preprocessed_doc: Any, results: Any) -> dict[str,
Any]:
        """
        Calculate SPC utilization metrics for verification manifest.

        Args:
            preprocessed_doc: PreprocessedDocument with chunk information
            results: Orchestrator execution results

        Returns:
            Dictionary with chunk metrics
        """
        if preprocessed_doc is None:
            return {}

        processing_mode = getattr(preprocessed_doc, 'processing_mode', 'flat')

        if processing_mode != 'chunked':
            return {
                "processing_mode": "flat",
                "note": "Document processed in flat mode (no chunk utilization)"
            }

        chunks = getattr(preprocessed_doc, 'chunks', [])
        chunk_graph = getattr(preprocessed_doc, 'chunk_graph', {})

        chunk_metrics = {
```

```
            "processing_mode": "chunked",
            "total_chunks": len(chunks),
            "chunk_types": {},
            "chunk_routing": {},
            "graph_metrics": {},
            "execution_savings": {}
        }

        # Count chunk types
        for chunk in chunks:
            chunk_type = getattr(chunk, 'chunk_type', 'unknown')
            chunk_metrics["chunk_types"][chunk_type] = \
                chunk_metrics["chunk_types"].get(chunk_type, 0) + 1

        # Calculate graph metrics if networkx available
        try:
            import networkx as nx

            if chunk_graph and isinstance(chunk_graph, dict):
                nodes = chunk_graph.get("nodes", [])
                edges = chunk_graph.get("edges", [])

                # Build networkx graph for analysis
                G = nx.DiGraph()
                for node in nodes:
                    node_id = node.get("id")
                    if node_id is not None:
                        G.add_node(node_id)

                for edge in edges:
                    source = edge.get("source")
                    target = edge.get("target")
                    if source is not None and target is not None:
                        G.add_edge(source, target)

                chunk_metrics["graph_metrics"] = {
                    "nodes": G.number_of_nodes(),
                    "edges": G.number_of_edges(),
                    "is_dag": nx.is_directed_acyclic_graph(G),
                     "is_connected": nx.is_weakly_connected(G) if G.number_of_nodes() > 0
else False,
                      "density": round(nx.density(G), 4) if G.number_of_nodes() > 0 else
0.0,
                }

                # Calculate diameter if connected
                if chunk_metrics["graph_metrics"]["is_connected"]:
                    try:
                                              chunk_metrics["graph_metrics"]["diameter"]  =
nx.diameter(G.to_undirected())
                    except Exception:
                        chunk_metrics["graph_metrics"]["diameter"] = -1
                else:
                    chunk_metrics["graph_metrics"]["diameter"] = -1
```

```
        except ImportError:
            chunk_metrics["graph_metrics"] = {
                "note": "NetworkX not available for graph analysis"
            }
        except Exception as e:
            chunk_metrics["graph_metrics"] = {
                "error": f"Graph analysis failed: {e!s}"
            }

        # Calculate execution savings
        # Use actual metrics from orchestrator if available
              if results and hasattr(results, '_execution_metrics') and 'phase_2' in
results._execution_metrics:
            metrics = results._execution_metrics['phase_2']
            chunk_metrics["execution_savings"] = {
                "chunk_executions": metrics['chunk_executions'],
                "full_doc_executions": metrics['full_doc_executions'],
                "total_possible_executions": metrics['total_possible_executions'],
                "actual_executions": metrics['actual_executions'],
                "savings_percent": round(metrics['savings_percent'], 2),
                "note": "Actual execution counts from orchestrator Phase 2"
            }
        elif results:
            # Fallback to estimation if real metrics not available
            total_possible_executions = 30 * len(chunks)  # 30 executors per chunk max
            # Assume chunk routing reduces executions by using type-specific executors
                    estimated_actual = len(chunks) * 10   # ~10 executors per chunk
(conservative)

            chunk_metrics["execution_savings"] = {
                "total_possible_executions": total_possible_executions,
                "estimated_actual_executions": estimated_actual,
                "estimated_savings_percent": round(
                    (1 - estimated_actual / max(total_possible_executions, 1)) * 100, 2
                ) if total_possible_executions > 0 else 0.0,
                   "note": "Estimated savings based on chunk-aware routing (orchestrator
metrics not available)"
            }

        return chunk_metrics

    def _calculate_signal_metrics(self, results: Any) -> dict[str, Any]:
        """
        Calculate signal utilization metrics for verification manifest.

        Args:
            results: Orchestrator execution results

        Returns:
            Dictionary with signal metrics
        """
        # Try to extract signal usage from results
        try:
```

```
            signal_metrics = {
                "enabled": True,
                "transport": "memory",
                "policy_areas_loaded": 10,
            }

            # Check if results have executor information
            if results and hasattr(results, 'executor_metadata'):
                # Count executors that used signals
                executors_with_signals = 0
                total_executors = 0

                for metadata in results.executor_metadata.values():
                    total_executors += 1
                    if metadata.get('signal_usage'):
                        executors_with_signals += 1

                signal_metrics["executors_using_signals"] = executors_with_signals
                signal_metrics["total_executors"] = total_executors

            # Default values if we can't extract from results
            if "executors_using_signals" not in signal_metrics:
                signal_metrics["executors_using_signals"] = 0
                signal_metrics["total_executors"] = 0
                    signal_metrics["note"] = "Signal infrastructure initialized, actual
usage not tracked in results"

            # Add signal pack versions
            signal_metrics["signal_versions"] = {
                f"PA{i:02d}": "1.0.0" for i in range(1, 11)
            }

            return signal_metrics

        except Exception as e:
            # If signal system not initialized, return minimal info
            return {
                "enabled": False,
                "note": f"Signal system not initialized: {e!s}"
            }

    def generate_verification_manifest(self, artifacts: list[str],
                                       artifact_hashes: dict[str, str],
                                       preprocessed_doc: Any = None,
                                       results: Any = None) -> Path:
        """
              Generate final verification manifest with SPC utilization metrics and
cryptographic integrity.

        Args:
            artifacts: List of artifact paths
            artifact_hashes: Dictionary mapping paths to SHA256 hashes
            preprocessed_doc: PreprocessedDocument (optional, for chunk metrics)
            results: Orchestrator results (optional, for execution metrics)
```

```
        Returns:
            Path to verification_manifest.json
        """
        end_time = datetime.utcnow().isoformat()

        # Calculate chunk utilization metrics
        chunk_metrics = self._calculate_chunk_metrics(preprocessed_doc, results)

        # Determine success based on strict criteria
        success = (
            self.phases_failed == 0 and
            self.phases_completed > 0 and
            len(self.errors) == 0 and
            len(artifacts) > 0
        )

        # Build manifest using VerificationManifestBuilder with HMAC integrity
        self.manifest_builder.set_success(success)
        self.manifest_builder.set_pipeline_hash(getattr(self, 'input_pdf_sha256', ''))

        # Add environment information
        self.manifest_builder.add_environment_info()

        # Add determinism information from seed registry
        seed_manifest = self.seed_registry.get_manifest_entry()
        self.manifest_builder.set_determinism_info(seed_manifest)

        # Add ingestion information
        if preprocessed_doc and hasattr(preprocessed_doc, 'metadata'):
            chunk_count = len(preprocessed_doc.metadata.get('chunks', []))
               text_length = len(preprocessed_doc.raw_text) if hasattr(preprocessed_doc,
'raw_text') else 0
                                sentence_count   =   len(preprocessed_doc.sentences)   if
hasattr(preprocessed_doc, 'sentences') else 0

            self.manifest_builder.add_ingestion_info({
                "method": "SPC",
                "chunk_count": chunk_count,
                "text_length": text_length,
                "sentence_count": sentence_count,
                "chunk_strategy": "semantic",
                "chunk_overlap": 50
            })

        # Add phase information
        self.manifest_builder.add_phase_info({
            "phase_name": "complete_pipeline",
            "status": "success" if success else "failed",
            "phases_completed": self.phases_completed,
            "phases_failed": self.phases_failed,
                                "duration_seconds":   (datetime.fromisoformat(end_time)   -
datetime.fromisoformat(self.start_time)).total_seconds()
        })
```

```
        # Add artifacts (including questionnaire as first-class artifact)
        for artifact_path, artifact_hash in artifact_hashes.items():
            self.manifest_builder.add_artifact(artifact_path, artifact_hash)

        # Add questionnaire as explicit artifact (SIN_CARRETA compliance)
        if hasattr(self, 'questionnaire_sha256'):
            self.manifest_builder.add_artifact(
                str(self.questionnaire_path),
                self.questionnaire_sha256
            )
            self.log_claim("artifact", "questionnaire",
                           "Questionnaire added to manifest",
                           {"path": str(self.questionnaire_path),
                            "hash": self.questionnaire_sha256})

        # Add SPC utilization metrics
        if chunk_metrics:
            self.manifest_builder.manifest_data["spc_utilization"] = chunk_metrics

        # Add legacy fields for backward compatibility
        self.manifest_builder.manifest_data.update({
            "execution_id": self.execution_id,
            "start_time": self.start_time,
            "end_time": end_time,
            "input_pdf_path": str(self.plan_pdf_path),
            "total_claims": len(self.claims),
            "errors": self.errors
        })

        # Add signal metrics to builder BEFORE building (fix use-before-define bug)
        signal_metrics = self._calculate_signal_metrics(results)
        if signal_metrics:
            self.manifest_builder.manifest_data["signals"] = signal_metrics

        # Build and save manifest with HMAC integrity
        manifest_path = self.artifacts_dir / "verification_manifest.json"
        manifest_json = self.manifest_builder.build(
                                            secret_key=os.environ.get("MANIFEST_SECRET_KEY",
"default-dev-key-change-in-production")
        )

        with open(manifest_path, 'w') as f:
            f.write(manifest_json)

        # Verify manifest integrity immediately
        manifest_dict = json.loads(manifest_json)
        is_valid, message = verify_manifest_integrity(
            manifest_dict,
                                            secret_key=os.environ.get("MANIFEST_SECRET_KEY",
"default-dev-key-change-in-production")
        )

        if not is_valid:
```

```
            self.log_claim("error", "verification_manifest",
                           f"Manifest integrity verification failed: {message}")
        else:
            self.log_claim("hash", "verification_manifest",
                           f"Manifest integrity verified: {message}",
                           {"file": str(manifest_path), "hmac_present": True})

        # Print verification banner
        if success and is_valid:
            print("\n" + "="*80)
            print("PIPELINE_VERIFIED=1")
            print(f"Manifest: {manifest_path}")
            print(f"HMAC: {manifest_dict.get('integrity_hmac', 'N/A')[:16]}...")
                 print(f"Phases: {self.phases_completed} completed, {self.phases_failed}
failed")
            print(f"Artifacts: {len(artifacts)}")
            print("="*80 + "\n")

        return manifest_path

    async def run(self) -> bool:
        """
        Execute the complete verified pipeline.

        Returns:
            True if pipeline succeeded, False otherwise
        """
        self.log_claim("start", "pipeline", "Starting verified pipeline execution")

        # Step 1: Verify input
        if not self.verify_input():
            self.generate_verification_manifest([], {})
            return False

        # Step 2: Run SPC ingestion (canonical phase-one)
        cpp = await self.run_spc_ingestion()
        if cpp is None:
            self.generate_verification_manifest([], {})
            return False

        # Step 3: Run CPP adapter
        preprocessed_doc = await self.run_cpp_adapter(cpp)
        if preprocessed_doc is None:
            self.generate_verification_manifest([], {})
            return False

        # Step 4: Run orchestrator
        results = await self.run_orchestrator(preprocessed_doc)
        if results is None:
            self.generate_verification_manifest([], {})
            return False

        # Step 5: Persist orchestrator metrics
        metrics_artifacts, metrics_hashes = self.persist_orchestrator_metrics()
```

```
        # Step 6: Save artifacts
        artifacts, artifact_hashes = self.save_artifacts(cpp, preprocessed_doc, results)

        # Merge metrics artifacts into main artifacts list
        artifacts.extend(metrics_artifacts)
        artifact_hashes.update(metrics_hashes)

        # Step 7: Generate verification manifest with chunk metrics
        manifest_path = self.generate_verification_manifest(
            artifacts, artifact_hashes, preprocessed_doc, results
        )

        self.log_claim("complete", "pipeline",
                       "Pipeline execution completed",
                       {
                           "success": self.phases_failed == 0,
                           "phases_completed": self.phases_completed,
                           "phases_failed": self.phases_failed,
                           "manifest_path": str(manifest_path)
                       })

        return self.phases_failed == 0


async def main():
    """Main entry point."""
    import argparse

    parser = argparse.ArgumentParser(
        description="Run verified policy pipeline with cryptographic verification"
    )
    parser.add_argument(
        "--plan",
        type=str,
        default="data/plans/Plan_1.pdf",
        help="Path to plan PDF (default: data/plans/Plan_1.pdf)"
    )
    parser.add_argument(
        "--artifacts-dir",
        type=str,
        default="artifacts/plan1",
        help="Directory for artifacts (default: artifacts/plan1)"
    )

    args = parser.parse_args()

    # Resolve paths
    plan_path = REPO_ROOT / args.plan
    artifacts_dir = REPO_ROOT / args.artifacts_dir

    print("=" * 80, flush=True)
    print("F.A.R.F.A.N VERIFIED POLICY PIPELINE RUNNER", flush=True)
    print("Framework for Advanced Retrieval of Administrativa Narratives", flush=True)
```

```
    print("=" * 80, flush=True)
    print(f"Plan: {plan_path}", flush=True)
    print(f"Artifacts: {artifacts_dir}", flush=True)
    print("=" * 80, flush=True)

    # Create and run pipeline
    runner = VerifiedPipelineRunner(plan_path, artifacts_dir)
    success = await runner.run()

    print("=" * 80, flush=True)
    if success:
        print("PIPELINE_VERIFIED=1", flush=True)
        print("Status: SUCCESS", flush=True)
    else:
        print("PIPELINE_VERIFIED=0", flush=True)
        print("Status: FAILED", flush=True)
    print("=" * 80, flush=True)

    sys.exit(0 if success else 1)


if __name__ == "__main__":
    asyncio.run(main())
```

scripts/sync_contract_group.py

```
#!/usr/bin/env python3
"""
SOTA Parallel Contract Orchestration System
============================================

State-of-the-art contract synchronization with:
- Parallel processing of all 30 groups maintaining positionality
- Cross-group pattern learning and transfer
- Intelligent repair propagation
- Real-time monitoring and adaptive strategies
"""

import json
import hashlib
import asyncio
import aiofiles
from pathlib import Path
from typing import Dict, List, Set, Optional, Any, Tuple, NamedTuple
from dataclasses import dataclass, field
from enum import Enum
import logging
from datetime import datetime, timezone
from collections import defaultdict, Counter
import numpy as np
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
import networkx as nx
from functools import lru_cache
import pickle

# Configure advanced logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - [%(levelname)s] - %(name)s - %(message)s',
    handlers=[
        logging. FileHandler('contract_orchestration.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)

# === Configuration ===
CONTRACTS_DIR                                                                    =
Path("src/canonic_phases/Phase_two/json_files_phase_two/executor_contracts/specialized")
CACHE_DIR = Path(". contract_cache")
CACHE_DIR.mkdir(exist_ok=True)

NUM_GROUPS = 30
NUM_POLICY_AREAS = 10
MAX_WORKERS = 10  # For parallel processing


# === Advanced Types ===
```

```
class PositionalEquivalence(NamedTuple):
    """Represents positional equivalence across groups"""
    dimension: int  # 1-3
    question_num: int  # 1-10
    group_id: int  # 0-29
    policy_areas: List[str]  # PA01-PA10
    question_ids: List[str]  # Q001, Q031, Q061...


@dataclass
class GroupProfile:
    """Profile of a contract group with learned patterns"""
    group_id:  int
    base_slot: str
    golden_contract: Optional[str] = None
    structural_signature: Optional[str] = None
    common_patterns: List[Dict] = field(default_factory=list)
    repair_strategies: List[str] = field(default_factory=list)
    confidence_score: float = 0.0
    contracts:  Dict[str, Any] = field(default_factory=dict)


@dataclass
class CrossGroupInsight:
    """Insights learned across groups"""
    pattern_type: str
    prevalence: float  # 0-1 across groups
    affected_groups: List[int]
    recommended_fix: Optional[Dict] = None
    confidence: float = 0.0


class RepairStrategy(Enum):
    """Advanced repair strategies"""
    GOLDEN_TRANSFER = "golden_transfer"  # Copy from golden contract
    PATTERN_INFERENCE = "pattern_inference"  # Infer from patterns
    CROSS_GROUP_LEARNING = "cross_group_learning"  # Learn from other groups
    STRUCTURAL_RECONSTRUCTION = "structural_reconstruction"  # Rebuild structure
    SEMANTIC_ALIGNMENT = "semantic_alignment"  # Align semantically
    GRAPH_OPTIMIZATION = "graph_optimization"  # Optimize dependency graph


# === Core Orchestrator ===

class SOTAContractOrchestrator:
    """State-of-the-art parallel contract orchestration system"""

    def __init__(self):
        self.groups: Dict[int, GroupProfile] = {}
        self.cross_group_insights: List[CrossGroupInsight] = []
        self.repair_cache: Dict[str, Any] = {}
        self.dependency_graph = nx.DiGraph()
        self._initialize_groups()
```

```
    def _initialize_groups(self):
        """Initialize all 30 groups with positional equivalence"""
        for group_id in range(NUM_GROUPS):
            dimension = (group_id // 10) + 1
            question_num = (group_id % 10) + 1
            base_slot = f"D{dimension}-Q{question_num}"

            self.groups[group_id] = GroupProfile(
                group_id=group_id,
                base_slot=base_slot
            )

    def get_positional_equivalence(self, group_id: int) -> PositionalEquivalence:
        """Get positional equivalence for a group"""
        dimension = (group_id // 10) + 1
        question_num = (group_id % 10) + 1
        base = group_id + 1

                    question_ids  =  [f"Q{base  +  (i  *  NUM_GROUPS):03d}"  for  i  in
range(NUM_POLICY_AREAS)]
        policy_areas = [f"PA{i+1:02d}" for i in range(NUM_POLICY_AREAS)]

        return PositionalEquivalence(
            dimension=dimension,
            question_num=question_num,
            group_id=group_id,
            policy_areas=policy_areas,
            question_ids=question_ids
        )

    async def orchestrate_all_groups(self,
                                     repair:  bool = True,
                                     parallel: bool = True) -> Dict[str, Any]:
        """Orchestrate verification and repair for all 30 groups"""
        logger.info("? Starting SOTA parallel orchestration for 30 groups")

        # Phase 1: Load and profile all groups
        await self._load_all_groups()

        # Phase 2: Identify golden contracts and patterns
        await self._identify_golden_contracts()

        # Phase 3: Cross-group learning
        self._learn_cross_group_patterns()

        # Phase 4: Parallel verification and repair
        if parallel:
            results = await self._parallel_process_groups(repair)
        else:
            results = await self._sequential_process_groups(repair)

        # Phase 5: Post-processing and optimization
        await self._optimize_results(results)
```

```
        # Phase 6: Generate comprehensive report
        report = self._generate_master_report(results)

        return report

    async def _load_all_groups(self):
        """Load all contracts organized by groups"""
        logger.info("? Loading all contracts...")

        async def load_contract(qid: str) -> Tuple[str, Optional[Dict]]:
            path = CONTRACTS_DIR / f"{qid}.v3.json"
            if not path.exists():
                return qid, None

            try:
                async with aiofiles.open(path, 'r') as f:
                    content = await f.read()
                    return qid, json.loads(content)
            except Exception as e:
                logger.error(f"Failed to load {qid}: {e}")
                return qid, None

        # Load all contracts in parallel
        tasks = []
        for group_id in range(NUM_GROUPS):
            pe = self.get_positional_equivalence(group_id)
            for qid in pe.question_ids:
                tasks.append(load_contract(qid))

        results = await asyncio.gather(*tasks)

        # Organize by groups
        for qid, contract in results:
            if contract:
                group_id = self._get_group_id_from_qid(qid)
                self.groups[group_id].contracts[qid] = contract

           logger.info(f"? Loaded {sum(len(g.contracts) for g in self.groups.values())}
contracts")

    def _get_group_id_from_qid(self, qid: str) -> int:
        """Extract group ID from question ID"""
        q_num = int(qid[1:])  # Q001 -> 1
        return (q_num - 1) % NUM_GROUPS

    async def _identify_golden_contracts(self):
        """Identify golden contract for each group using sophisticated metrics"""
        logger.info("? Identifying golden contracts...")

        for group_id, profile in self.groups.items():
            if not profile.contracts:
                continue
```

```
        scores = {}
        for qid, contract in profile.contracts.items():
            score = self._calculate_contract_quality_score(contract)
            scores[qid] = score

        if scores:
            golden = max(scores, key=scores. get)
            profile.golden_contract = golden
            profile.confidence_score = scores[golden] / 100.0

            # Generate structural signature
            profile.structural_signature = self._generate_structural_signature(
                profile.contracts[golden]
            )

            logger.info(f"Group {group_id} ({profile.base_slot}): "
                      f"Golden={golden} (score={scores[golden]:.1f})")

def _calculate_contract_quality_score(self, contract: Dict) -> float:
    """Calculate sophisticated quality score for a contract"""
    score = 0.0

    # Check critical structures (40 points)
    critical_structures = [
        "method_binding. execution_sequence",
        "method_outputs",
        "evidence_structure_post_nexus",
        "human_answer_structure. evidence_structure_schema",
        "human_answer_structure.concrete_example"
    ]

    for structure in critical_structures:
        if self._deep_get(contract, structure):
            score += 8

    # Check method documentation (20 points)
    methods = self._deep_get(contract, "method_binding.methods") or []
    if len(methods) >= 17:
        score += 10
    if contract.get("method_outputs"):
        score += 10

    # Check human answer structure completeness (20 points)
    has = contract.get("human_answer_structure", {})
    if has.get("evidence_structure_schema"):
        schema = has["evidence_structure_schema"]. get("properties", {})
        if len(schema) >= 10:
            score += 10
    if has.get("concrete_example"):
        score += 10

    # Check evidence assembly sophistication (10 points)
    if self._deep_get(contract, "evidence_assembly.class_name") == "EvidenceNexus":
        score += 10
```

```
        # Check pattern diversity (10 points)
        patterns = self._deep_get(contract, "question_context.patterns") or []
        categories = set(p.get("category") for p in patterns if p.get("category"))
        if len(patterns) >= 10 and len(categories) >= 3:
            score += 10

        return score

    def _generate_structural_signature(self, contract:  Dict) -> str:
        """Generate unique structural signature for a contract"""
        signature_parts = []

        # Include key structural elements
                    signature_parts.append(f"methods:{len(contract.get('method_binding',
{}).get('methods', []))}")
                    signature_parts.append(f"has_exec_seq:{bool(self._deep_get(contract,
'method_binding.execution_sequence'))}")
        signature_parts. append(f"has_outputs:{bool(contract.get('method_outputs'))}")
                            signature_parts.append(f"has_nexus:{self._deep_get(contract,
'evidence_assembly.class_name') == 'EvidenceNexus'}")

signature_parts.append(f"has_human_struct:{bool(contract.get('human_answer_structure'))}
")

        signature_str = "|".join(signature_parts)
        return hashlib. md5(signature_str.encode()).hexdigest()[:16]

    def _learn_cross_group_patterns(self):
        """Learn patterns across all groups for intelligent repair"""
        logger.info("? Learning cross-group patterns...")

        # Analyze common issues across groups
        issue_patterns = defaultdict(list)

        for group_id, profile in self.groups.items():
            for qid, contract in profile.contracts.items():
                issues = self._quick_scan_issues(contract)
                for issue in issues:
                    issue_patterns[issue]. append(group_id)

        # Generate insights
        for issue_type, affected_groups in issue_patterns.items():
            if len(affected_groups) >= 5:  # Pattern appears in 5+ groups
                insight = CrossGroupInsight(
                    pattern_type=issue_type,
                    prevalence=len(affected_groups) / NUM_GROUPS,
                    affected_groups=affected_groups,
                    confidence=min(0.9, len(affected_groups) / 10)
                )
                self.cross_group_insights.append(insight)

               logger.info(f"?  Discovered  {len(self.cross_group_insights)}  cross-group
patterns")
```

```
    def _quick_scan_issues(self, contract: Dict) -> List[str]:
        """Quick scan for common issues"""
        issues = []

        if not self._deep_get(contract, "method_binding.execution_sequence"):
            issues.append("missing_execution_sequence")

        if not contract.get("method_outputs"):
            issues.append("missing_method_outputs")

                                                if      not      self._deep_get(contract,
"human_answer_structure.evidence_structure_schema"):
            issues.append("incomplete_human_structure")

        # Check identity-schema mismatch
        identity = contract.get("identity", {})
         schema_props = self._deep_get(contract, "output_contract.schema.properties") or
{}

        for field in ["dimension_id", "cluster_id"]:
            if identity.get(field) != schema_props.get(field, {}).get("const"):
                issues.append(f"mismatch_{field}")

        return issues

    async def _parallel_process_groups(self, repair: bool) -> Dict[int, Dict]:
        """Process all groups in parallel"""
            logger.info(f"? Processing {NUM_GROUPS} groups in parallel (max workers:
{MAX_WORKERS})")

        results = {}

        async def process_group(group_id: int) -> Tuple[int, Dict]:
            profile = self.groups[group_id]
            logger.info(f"Processing group {group_id} ({profile.base_slot})...")

            group_result = {
                "group_id": group_id,
                "base_slot": profile.base_slot,
                "contracts_processed": len(profile.contracts),
                "golden_contract": profile.golden_contract,
                "issues_found": {},
                "repairs_applied": {},
                "verification_results": {}
            }

            # Process each contract in the group
            for qid, contract in profile.contracts.items():
                # Verify
                verifier = AdvancedContractVerifier()
                issues = verifier.verify_contract(contract, qid)
                group_result["issues_found"][qid] = len(issues)
```

```
                # Repair if requested
                if repair and issues:
                    repairer = IntelligentContractRepairer(
                        golden_contract=profile.contracts.get(profile.golden_contract),
                        cross_group_insights=self.cross_group_insights
                    )

                                                repaired_contract,  repair_result  =  await
repairer.repair_contract_async(
                        contract, issues, qid
                    )

                    if repair_result.success:
                        # Save repaired contract
                        await self._save_contract(qid, repaired_contract)
                                                    group_result["repairs_applied"][qid]  =
len(repair_result.issues_fixed)

                    # Re-verify
                    remaining_issues = verifier.verify_contract(repaired_contract, qid)
                    group_result["verification_results"][qid] = {
                        "initial_issues": len(issues),
                        "remaining_issues": len(remaining_issues),
                        "fixed":  len(issues) - len(remaining_issues)
                    }

            return group_id, group_result

        # Create tasks for all groups
        tasks = [process_group(gid) for gid in range(NUM_GROUPS)]

        # Process with limited concurrency
        sem = asyncio.Semaphore(MAX_WORKERS)

        async def bounded_process(group_id: int) -> Tuple[int, Dict]:
            async with sem:
                return await process_group(group_id)

        bounded_tasks = [bounded_process(gid) for gid in range(NUM_GROUPS)]
        group_results = await asyncio.gather(*bounded_tasks)

        # Organize results
        for group_id, result in group_results:
            results[group_id] = result

        return results

    async def _sequential_process_groups(self, repair: bool) -> Dict[int, Dict]:
        """Process groups sequentially (fallback)"""
        results = {}

        for group_id in range(NUM_GROUPS):
            result = await self._parallel_process_groups(repair)
            results. update(result)
```

```
        return results

    async def _optimize_results(self, results: Dict[int, Dict]):
        """Post-process and optimize results"""
        logger.info("? Optimizing results...")

        # Build dependency graph for cross-group optimization
        for group_id in range(NUM_GROUPS):
            pe = self.get_positional_equivalence(group_id)

            # Add edges based on positional relationships
            # Same dimension, adjacent questions
            if pe.question_num < 10:
                next_group = group_id + 1
                if next_group < NUM_GROUPS and (next_group // 10) == (group_id // 10):
                    self.dependency_graph. add_edge(group_id, next_group)

            # Same question, different dimensions
            for dim in range(3):
                other_group = (dim * 10) + pe.question_num - 1
                if other_group != group_id and 0 <= other_group < NUM_GROUPS:
                    self.dependency_graph.add_edge(group_id, other_group, weight=0.5)

        # Propagate successful repairs across related groups
        for group_id, result in results.items():
            if result. get("repairs_applied"):
                # Find related groups
                related = list(self.dependency_graph.neighbors(group_id))

                for related_group in related:
                    # Apply similar repairs if applicable
                      logger.debug(f"Considering repair propagation from group {group_id}
to {related_group}")

    async def _save_contract(self, qid: str, contract: Dict):
        """Save contract to disk"""
        path = CONTRACTS_DIR / f"{qid}.v3.json"

        # Update timestamp and hash
        contract["identity"]["updated_at"] = datetime.now(timezone.utc).isoformat()
        contract["identity"]["contract_hash"] = self._compute_contract_hash(contract)

        async with aiofiles.open(path, 'w') as f:
            await f.write(json.dumps(contract, indent=2, ensure_ascii=False))

    def _compute_contract_hash(self, contract: Dict) -> str:
        """Compute contract hash"""
        contract_copy = json.loads(json.dumps(contract))
        if "identity" in contract_copy:
            contract_copy["identity"].pop("contract_hash", None)
            contract_copy["identity"].pop("updated_at", None)

        content = json.dumps(contract_copy, sort_keys=True, ensure_ascii=False)
```

```
        return hashlib.sha256(content.encode()).hexdigest()

    def _generate_master_report(self, results: Dict[int, Dict]) -> Dict:
        """Generate comprehensive master report"""
        logger.info("? Generating master report...")

        total_contracts = sum(r["contracts_processed"] for r in results.values())
        total_issues = sum(sum(r["issues_found"].values()) for r in results.values())
                  total_repairs  =  sum(sum(r["repairs_applied"].values())  for  r  in
results.values())

        # Calculate success metrics by dimension
        dimension_stats = defaultdict(lambda: {"groups": 0, "issues": 0, "repairs": 0})

        for group_id, result in results.items():
            dimension = (group_id // 10) + 1
            dimension_stats[dimension]["groups"] += 1
            dimension_stats[dimension]["issues"] += sum(result["issues_found"].values())
                                                dimension_stats[dimension]["repairs"]   +=
sum(result["repairs_applied"].values())

        report = {
            "timestamp": datetime.now(timezone.utc).isoformat(),
            "orchestration_mode": "SOTA_PARALLEL",
            "summary": {
                "total_groups": NUM_GROUPS,
                "total_contracts": total_contracts,
                "total_issues_found": total_issues,
                "total_repairs_applied": total_repairs,
                       "repair_success_rate": (total_repairs / total_issues * 100) if
total_issues else 0,
                "cross_group_insights": len(self. cross_group_insights)
            },
            "dimension_statistics": dict(dimension_stats),
            "group_results": results,
            "cross_group_insights": [
                {
                    "pattern":  insight.pattern_type,
                    "prevalence": f"{insight.prevalence:.1%}",
                    "affected_groups": insight.affected_groups,
                    "confidence": insight.confidence
                }
                for insight in self.cross_group_insights
            ],
            "golden_contracts": {
                gid: prof.golden_contract
                for gid, prof in self.groups.items()
                if prof.golden_contract
            }
        }

        # Save report
        report_path = Path("master_orchestration_report.json")
        with open(report_path, 'w') as f:
```

```
            json.dump(report, f, indent=2)

        # Generate HTML dashboard
        self._generate_html_dashboard(report)

        logger.info(f"? Master report saved to {report_path}")

        return report

    def _generate_html_dashboard(self, report: Dict):
        """Generate interactive HTML dashboard"""
        html = f"""
<! DOCTYPE html>
<html>
<head>
    <title>SOTA Contract Orchestration Dashboard</title>
    <style>
        * {{ margin: 0; padding:  0; box-sizing: border-box; }}
        body {{
                    font-family:  -apple-system,  BlinkMacSystemFont,  'Segoe  UI',  Roboto,
sans-serif;
            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
            min-height: 100vh;
            padding: 20px;
        }}
        .container {{
            max-width: 1600px;
            margin: 0 auto;
        }}
        .header {{
            background: rgba(255,255,255,0.95);
            border-radius: 20px;
            padding: 30px;
            box-shadow: 0 20px 60px rgba(0,0,0,0.3);
            margin-bottom: 30px;
        }}
        h1 {{
            font-size: 2.5em;
            background: linear-gradient(135deg, #667eea, #764ba2);
            -webkit-background-clip: text;
            -webkit-text-fill-color: transparent;
            margin-bottom: 10px;
        }}
        .timestamp {{
            color: #666;
            font-size: 0.9em;
        }}
        .metrics-grid {{
            display: grid;
            grid-template-columns:  repeat(auto-fit, minmax(250px, 1fr));
            gap: 20px;
            margin:  30px 0;
        }}
        .metric-card {{
```

```
    background: rgba(255,255,255,0.95);
    border-radius: 15px;
    padding: 25px;
    box-shadow: 0 10px 30px rgba(0,0,0,0.2);
    transition: transform 0.3s;
}}
.metric-card:hover {{
    transform:  translateY(-5px);
}}
.metric-value {{
    font-size: 3em;
    font-weight:  bold;
    background: linear-gradient(135deg, #667eea, #764ba2);
    -webkit-background-clip: text;
    -webkit-text-fill-color: transparent;
}}
.metric-label {{
    color: #666;
    font-size:  0.9em;
    text-transform: uppercase;
    letter-spacing: 1px;
    margin-top: 10px;
}}
.dimension-grid {{
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    gap: 20px;
    margin: 30px 0;
}}
.dimension-card {{
    background: rgba(255,255,255,0.95);
    border-radius: 15px;
    padding: 20px;
    box-shadow: 0 10px 30px rgba(0,0,0,0.2);
}}
. dimension-title {{
    font-size: 1.3em;
    font-weight:  bold;
    color: #333;
    margin-bottom: 15px;
    border-bottom: 2px solid #667eea;
    padding-bottom: 10px;
}}
.group-matrix {{
    display: grid;
    grid-template-columns: repeat(10, 1fr);
    gap: 5px;
    margin: 30px 0;
    background: rgba(255,255,255,0.95);
    padding: 20px;
    border-radius:  15px;
    box-shadow: 0 10px 30px rgba(0,0,0,0.2);
}}
.group-cell {{
```

```
            aspect-ratio: 1;
            border-radius: 8px;
            display: flex;
            align-items: center;
            justify-content: center;
            font-weight: bold;
            color: white;
            font-size: 0.9em;
            cursor: pointer;
            transition: all 0.3s;
        }}
        .group-cell:hover {{
            transform: scale(1.1);
            z-index: 10;
        }}
        .success {{ background: linear-gradient(135deg, #00c851, #00ff00); }}
        .partial {{ background: linear-gradient(135deg, #ffbb33, #FF8800); }}
        .failed {{ background: linear-gradient(135deg, #ff4444, #CC0000); }}
        .insights {{
            background: rgba(255,255,255,0.95);
            border-radius: 15px;
            padding: 25px;
            box-shadow: 0 10px 30px rgba(0,0,0,0.2);
            margin:  30px 0;
        }}
        .insight-item {{
            padding: 15px;
            margin: 10px 0;
            background: #f8f9fa;
            border-left: 4px solid #667eea;
            border-radius: 8px;
        }}
        . progress-bar {{
            width: 100%;
            height: 30px;
            background: #e0e0e0;
            border-radius: 15px;
            overflow: hidden;
            margin: 20px 0;
        }}
        .progress-fill {{
            height: 100%;
            background: linear-gradient(90deg, #667eea, #764ba2);
            border-radius: 15px;
            display: flex;
            align-items: center;
            justify-content: center;
            color: white;
            font-weight: bold;
            transition: width 0.5s ease;
        }}
    </style>
</head>
<body>
```

```
    <div class="container">
        <div class="header">
            <h1>? SOTA Contract Orchestration Dashboard</h1>
            <div class="timestamp">Generated: {report['timestamp']}</div>
        </div>

        <div class="metrics-grid">
            <div class="metric-card">
                <div class="metric-value">{report['summary']['total_groups']}</div>
                <div class="metric-label">Groups Processed</div>
            </div>
            <div class="metric-card">
                <div class="metric-value">{report['summary']['total_contracts']}</div>
                <div class="metric-label">Contracts Analyzed</div>
            </div>
            <div class="metric-card">
                                                                                <div
class="metric-value">{report['summary']['total_issues_found']}</div>
                <div class="metric-label">Issues Found</div>
            </div>
            <div class="metric-card">
                                                                                <div
class="metric-value">{report['summary']['total_repairs_applied']}</div>
                <div class="metric-label">Repairs Applied</div>
            </div>
            <div class="metric-card">
                                                                                <div
class="metric-value">{report['summary']['repair_success_rate']:.1f}%</div>
                <div class="metric-label">Success Rate</div>
            </div>
            <div class="metric-card">
                                                                                <div
class="metric-value">{report['summary']['cross_group_insights']}</div>
                <div class="metric-label">Insights Discovered</div>
            </div>
        </div>

        <div class="progress-bar">
                                            <div    class="progress-fill"    style="width:
{report['summary']['repair_success_rate']}%">
                {report['summary']['repair_success_rate']:.1f}% Complete
            </div>
        </div>

        <div class="dimension-grid">
"""

        # Add dimension statistics
        for dim, stats in report['dimension_statistics'].items():
             repair_rate = (stats['repairs'] / stats['issues'] * 100) if stats['issues']
else 100
            html += f"""
            <div class="dimension-card">
                <div class="dimension-title">Dimension {dim}</div>
```

```
                <p>Groups: {stats['groups']}</p>
                <p>Issues: {stats['issues']}</p>
                <p>Repairs: {stats['repairs']}</p>
                <p>Success:  {repair_rate:.1f}%</p>
            </div>
"""

        html += """
        </div>

        <h2 style="color: white; margin:  20px 0;">Group Matrix (30 Groups)</h2>
        <div class="group-matrix">
"""

        # Add group matrix visualization
        for group_id in range(NUM_GROUPS):
            result = report['group_results'].get(group_id, {})
            issues = sum(result.get('issues_found', {}).values())
            repairs = sum(result.get('repairs_applied', {}).values())

            if issues == 0:
                status_class = "success"
            elif repairs >= issues * 0.8:
                status_class = "success"
            elif repairs >= issues * 0.5:
                status_class = "partial"
            else:
                status_class = "failed"

            dimension = (group_id // 10) + 1
            question = (group_id % 10) + 1

            html += f"""
                    <div  class="group-cell  {status_class}"  title="Group  {group_id}:
D{dimension}-Q{question}">
                D{dimension}-Q{question}
            </div>
"""

        html += """
        </div>

        <div class="insights">
            <h2>? Cross-Group Insights</h2>
"""

        # Add insights
        for insight in report['cross_group_insights'][:10]:  # Top 10 insights
            html += f"""
            <div class="insight-item">
                <strong>{insight['pattern']}</strong><br>
                Prevalence:  {insight['prevalence']}<br>
                Confidence: {insight['confidence']:.1f}<br>
                Affects {len(insight['affected_groups'])} groups
```

```
            </div>
"""

        html += """
        </div>
    </div>

    <script>
        // Add interactive features
        document.querySelectorAll('.group-cell').forEach(cell => {
            cell.addEventListener('click', function() {
                alert('Group details:  ' + this.title);
            });
        });
    </script>
</body>
</html>
"""

        dashboard_path = Path("orchestration_dashboard.html")
        with open(dashboard_path, 'w') as f:
            f. write(html)

        logger.info(f"? Dashboard saved to {dashboard_path}")

    def _deep_get(self, obj:  Dict, path: str) -> Any:
        """Get nested value from dict"""
        keys = path.split('.')
        current = obj
        for key in keys:
            if isinstance(current, dict):
                current = current.get(key)
                if current is None:
                    return None
            else:
                return None
        return current


# === Advanced Verifier ===

class AdvancedContractVerifier:
    """Advanced contract verification with pattern learning"""

    def verify_contract(self, contract: Dict, qid: str) -> List[Dict]:
        """Verify contract and return issues"""
        issues = []

        # Run verification checks
        self._check_identity_consistency(contract, qid, issues)
        self._check_method_evidence_alignment(contract, qid, issues)
        self._check_human_answer_structure(contract, qid, issues)

        return issues
```

```
    def _check_identity_consistency(self, contract: Dict, qid: str, issues: List):
        """Check identity field consistency"""
        identity = contract. get("identity", {})
                    schema_props   =   contract.get("output_contract",   {}).get("schema",
{}).get("properties", {})

        for field in ["dimension_id", "cluster_id", "question_global"]:
            identity_value = identity.get(field)
            schema_value = schema_props.get(field, {}).get("const")

            if identity_value != schema_value:
                issues.append({
                    "type": f"identity_mismatch_{field}",
                    "severity": "CRITICAL",
                    "field": field,
                    "identity_value": identity_value,
                    "schema_value": schema_value,
                    "fixable": True
                })

    def _check_method_evidence_alignment(self, contract: Dict, qid: str, issues: List):
        """Check method-evidence alignment"""
        methods = contract.get("method_binding", {}).get("methods", [])
        provides = {m.get("provides") for m in methods if m.get("provides")}

        assembly_rules = contract.get("evidence_assembly", {}).get("assembly_rules", [])

        for rule in assembly_rules:
            for source in rule.get("sources", []):
                if not source.startswith("*. "):
                     base = source.split(". ")[0] + "." + source.split(".")[1] if "." in
source else source
                    if base not in provides:
                        issues.append({
                            "type": "unmapped_source",
                            "severity": "ERROR",
                            "source": source,
                            "fixable":  True
                        })

    def _check_human_answer_structure(self, contract: Dict, qid: str, issues: List):
        """Check human answer structure"""
        has = contract.get("human_answer_structure", {})

        if not has.get("evidence_structure_schema"):
            issues.append({
                "type":  "missing_evidence_schema",
                "severity": "ERROR",
                "fixable": True
            })


# === Intelligent Repairer ===
```

```
class IntelligentContractRepairer:
    """Intelligent contract repair with cross-group learning"""

    def __init__(self, golden_contract: Optional[Dict] = None,
                 cross_group_insights: List[CrossGroupInsight] = None):
        self.golden_contract = golden_contract
        self.cross_group_insights = cross_group_insights or []

    async def repair_contract_async(self, contract: Dict, issues: List[Dict],
                                    qid: str) -> Tuple[Dict, Any]:
        """Repair contract using intelligent strategies"""
        repairs_applied = []

        for issue in issues:
            if issue. get("fixable"):
                strategy = self._select_repair_strategy(issue)

                if strategy == RepairStrategy.GOLDEN_TRANSFER and self.golden_contract:
                    contract = self._apply_golden_transfer(contract, issue)
                    repairs_applied.append(f"{issue['type']} via golden transfer")

                elif strategy == RepairStrategy.PATTERN_INFERENCE:
                    contract = self._apply_pattern_inference(contract, issue)
                    repairs_applied.append(f"{issue['type']} via pattern inference")

                elif strategy == RepairStrategy.CROSS_GROUP_LEARNING:
                    contract = self._apply_cross_group_learning(contract, issue)
                    repairs_applied.append(f"{issue['type']} via cross-group learning")

        # Return mock repair result
        from collections import namedtuple
        RepairResult = namedtuple('RepairResult', ['success', 'issues_fixed'])

        return contract, RepairResult(
            success=len(repairs_applied) > 0,
            issues_fixed=repairs_applied
        )

    def _select_repair_strategy(self, issue: Dict) -> RepairStrategy:
        """Select optimal repair strategy for issue"""
        if self.golden_contract and issue['type']. startswith('identity_mismatch'):
            return RepairStrategy. GOLDEN_TRANSFER

        # Check if cross-group insights apply
        for insight in self.cross_group_insights:
            if insight.pattern_type == issue['type'] and insight.confidence > 0.7:
                return RepairStrategy.CROSS_GROUP_LEARNING

        return RepairStrategy.PATTERN_INFERENCE

    def _apply_golden_transfer(self, contract: Dict, issue: Dict) -> Dict:
        """Apply fix from golden contract"""
        if issue['type']. startswith('identity_mismatch_'):
```

```
            field = issue['field']
                  contract['output_contract']['schema']['properties'][field]['const'] =
issue['identity_value']

        return contract

    def _apply_pattern_inference(self, contract:  Dict, issue: Dict) -> Dict:
        """Apply fix based on pattern inference"""
        # Implement pattern-based fixes
        return contract

    def _apply_cross_group_learning(self, contract: Dict, issue: Dict) -> Dict:
        """Apply fix based on cross-group learning"""
        # Implement cross-group learning fixes
        return contract


# === Main Execution ===

async def main():
    """Main execution entry point"""
    import argparse

    parser = argparse.ArgumentParser(description="SOTA Parallel Contract Orchestrator")
    parser.add_argument("--repair", action="store_true", help="Enable repair mode")
        parser.add_argument("--parallel",  action="store_true",  default=True,  help="Use
parallel processing")
       parser.add_argument("--workers",  type=int,  default=10,  help="Number  of  parallel
workers")

    args = parser.parse_args()

    print("="*80)
    print("? SOTA PARALLEL CONTRACT ORCHESTRATOR")
    print("="*80)
    print(f"Mode: {'REPAIR' if args. repair else 'VERIFY'}")
    print(f"Processing:  PARALLEL with {args.workers} workers")
    print(f"Groups: All 30 groups maintaining positional equivalence")
    print("="*80)

    # Initialize orchestrator
    orchestrator = SOTAContractOrchestrator()

    # Run orchestration
    report = await orchestrator.orchestrate_all_groups(
        repair=args.repair,
        parallel=args.parallel
    )

    # Print summary
    print("\n" + "="*80)
    print("? ORCHESTRATION COMPLETE")
    print("="*80)
    print(f"? Groups processed: {report['summary']['total_groups']}")
```

```
    print(f"? Contracts analyzed: {report['summary']['total_contracts']}")
    print(f"? Issues found: {report['summary']['total_issues_found']}")
    print(f"? Repairs applied: {report['summary']['total_repairs_applied']}")
    print(f"? Success rate: {report['summary']['repair_success_rate']:.1f}%")
    print(f"? Cross-group insights: {report['summary']['cross_group_insights']}")
    print("\n? Reports saved:")
    print("  ? master_orchestration_report.json")
    print("  ? orchestration_dashboard.html")
    print("="*80)

    return 0 if report['summary']['repair_success_rate'] > 80 else 1


if __name__ == "__main__":
    import sys
    sys.exit(asyncio.run(main()))
```

scripts/validate_contract_equivalence.py

```
#!/usr/bin/env python3
"""
Contract Equivalence Validator
==============================

Validates that contracts at equivalent positions (same base_slot across 10 policy areas)
maintain structural invariants while allowing expected variations.

Matrix Structure:
- 30 base slots (D1-Q1 through D6-Q5)
- 10 policy areas (PA01-PA10)
- 300 total contracts

Equivalence Groups:
- Group 0: Q001, Q031, Q061, Q091, Q121, Q151, Q181, Q211, Q241, Q271 ? all D1-Q1
- Group 1: Q002, Q032, Q062, Q092, Q122, Q152, Q182, Q212, Q242, Q272 ? all D1-Q2
- ...
- Group 29: Q030, Q060, Q090, Q120, Q150, Q180, Q210, Q240, Q270, Q300 ? all D6-Q5

Usage:
    python scripts/validate_contract_equivalence.py [--fix] [--group N] [--verbose]
"""

import argparse
import json
import hashlib
from pathlib import Path
from dataclasses import dataclass, field
from typing import Any
from collections import defaultdict


# === Configuration ===

CONTRACTS_DIR               =               Path(__file__).parent.parent               /
"src/canonic_phases/Phase_two/json_files_phase_two/executor_contracts/specialized"
NUM_BASE_SLOTS = 30
NUM_POLICY_AREAS = 10


# === Data Classes ===

@dataclass
class ValidationIssue:
    """Single validation issue found."""
    severity: str  # "ERROR", "WARNING", "INFO"
    group_id: int
    question_ids: list[str]
    component: str
    message: str
    details: dict[str, Any] = field(default_factory=dict)
```

```
@dataclass
class GroupValidationResult:
    """Validation result for one equivalence group."""
    group_id: int
    base_slot: str
    contracts_loaded: int
    contracts_expected: int
    issues: list[ValidationIssue] = field(default_factory=list)

    @property
    def is_valid(self) -> bool:
        return not any(i.severity == "ERROR" for i in self.issues)


# === Core Validation Logic ===

def load_contract(path: Path) -> dict[str, Any] | None:
    """Load a single contract JSON file."""
    try:
        with open(path, "r", encoding="utf-8") as f:
            return json.load(f)
    except (json.JSONDecodeError, FileNotFoundError) as e:
        print(f"  ??  Failed to load {path.name}: {e}")
        return None


def get_question_number(question_id: str) -> int:
    """Extract numeric part from question ID (e.g., 'Q001' -> 1)."""
    return int(question_id[1:])


def get_group_id(question_number: int) -> int:
    """Compute equivalence group from question number."""
    return (question_number - 1) % NUM_BASE_SLOTS


def get_expected_question_numbers_for_group(group_id: int) -> list[int]:
    """Get all question numbers that belong to a group."""
    return [group_id + 1 + (i * NUM_BASE_SLOTS) for i in range(NUM_POLICY_AREAS)]


def compute_structure_hash(obj: Any, ignore_keys: set[str] | None = None) -> str:
    """Compute hash of object structure, ignoring specified keys."""
    ignore_keys = ignore_keys or set()

    def _normalize(o: Any) -> Any:
        if isinstance(o, dict):
                    return {k: _normalize(v) for k, v in sorted(o.items()) if k not in
ignore_keys}
        elif isinstance(o, list):
            return [_normalize(item) for item in o]
        else:
            return o
```

```
    normalized = _normalize(obj)
                                        return         hashlib.sha256(json.dumps(normalized,
sort_keys=True).encode()).hexdigest()[:16]


def validate_invariant_section(
    contracts: dict[str, dict],
    section_path: str,
    group_id: int,
    ignore_keys: set[str] | None = None
) -> list[ValidationIssue]:
    """
    Validate that a section is structurally identical across all contracts in group.

    Args:
        contracts: Dict of question_id -> contract data
        section_path: Dot-separated path to section (e.g., "method_binding.methods")
        group_id: Equivalence group ID
        ignore_keys: Keys to ignore in comparison
    """
    issues = []

    def get_nested(d: dict, path: str) -> Any:
        for key in path.split("."):
            if isinstance(d, dict):
                d = d.get(key, {})
            else:
                return None
        return d

    hashes = {}
    for qid, contract in contracts.items():
        section = get_nested(contract, section_path)
        if section:
            h = compute_structure_hash(section, ignore_keys)
            hashes[qid] = h

    unique_hashes = set(hashes.values())
    if len(unique_hashes) > 1:
        # Find which contracts differ
        hash_groups = defaultdict(list)
        for qid, h in hashes.items():
            hash_groups[h].append(qid)

        # Report the minority groups as issues
        sorted_groups = sorted(hash_groups.items(), key=lambda x: -len(x[1]))
        majority_hash, majority_qids = sorted_groups[0]

        for h, qids in sorted_groups[1:]:
            issues.append(ValidationIssue(
                severity="ERROR",
                group_id=group_id,
                question_ids=qids,
```

```
                component=section_path,
                            message=f"Section  '{section_path}'  differs  from  majority
({len(majority_qids)} contracts)",
                details={
                    "divergent_contracts": qids,
                    "majority_contracts": majority_qids[:3],  # Sample
                    "hash_divergent": h,
                    "hash_majority": majority_hash
                }
            ))

    return issues


def validate_identity_consistency(
    contracts: dict[str, dict],
    group_id: int
) -> list[ValidationIssue]:
    """Validate identity block consistency within group."""
    issues = []

    base_slots = set()
    for qid, contract in contracts.items():
        identity = contract.get("identity", {})
        base_slots.add(identity.get("base_slot"))

    if len(base_slots) > 1:
        issues.append(ValidationIssue(
            severity="ERROR",
            group_id=group_id,
            question_ids=list(contracts.keys()),
            component="identity.base_slot",
            message=f"Inconsistent base_slot within group: {base_slots}",
            details={"found_slots": list(base_slots)}
        ))

    return issues


def validate_output_contract_consts(
    contracts: dict[str, dict],
    group_id: int
) -> list[ValidationIssue]:
    """
    Validate that output_contract.schema.properties uses correct const values
    matching the contract's identity block.
    """
    issues = []

    for qid, contract in contracts.items():
        identity = contract.get("identity", {})
                output_schema  =  contract.get("output_contract",  {}).get("schema",
{}).get("properties", {})
```

```
        # Check question_id const
        schema_qid = output_schema.get("question_id", {}).get("const")
        if schema_qid and schema_qid != identity.get("question_id"):
            issues.append(ValidationIssue(
                severity="ERROR",
                group_id=group_id,
                question_ids=[qid],
                component="output_contract.schema.properties.question_id",
                      message=f"Hardcoded wrong question_id: schema has '{schema_qid}',
identity has '{identity.get('question_id')}'",
                details={
                    "schema_value": schema_qid,
                    "identity_value": identity.get("question_id")
                }
            ))

        # Check question_global const
        schema_global = output_schema.get("question_global", {}).get("const")
        if schema_global and schema_global != identity.get("question_global"):
            issues.append(ValidationIssue(
                severity="ERROR",
                group_id=group_id,
                question_ids=[qid],
                component="output_contract.schema.properties.question_global",
                  message=f"Hardcoded wrong question_global: schema has {schema_global},
identity has {identity.get('question_global')}",
                details={
                    "schema_value": schema_global,
                    "identity_value": identity.get("question_global")
                }
            ))

        # Check policy_area_id const
        schema_pa = output_schema.get("policy_area_id", {}).get("const")
        if schema_pa and schema_pa != identity.get("policy_area_id"):
            issues.append(ValidationIssue(
                severity="ERROR",
                group_id=group_id,
                question_ids=[qid],
                component="output_contract.schema.properties.policy_area_id",
                     message=f"Hardcoded wrong policy_area_id: schema has '{schema_pa}',
identity has '{identity.get('policy_area_id')}'",
                details={
                    "schema_value": schema_pa,
                    "identity_value": identity.get("policy_area_id")
                }
            ))

    return issues


def validate_method_binding_invariants(
    contracts: dict[str, dict],
    group_id: int
```

```
) -> list[ValidationIssue]:
    """Validate method_binding is identical across group."""
    return validate_invariant_section(
        contracts,
        "method_binding",
        group_id,
        ignore_keys={"note"}  # Notes can vary
    )


def validate_executor_binding_invariants(
    contracts: dict[str, dict],
    group_id: int
) -> list[ValidationIssue]:
    """Validate executor_binding is identical across group."""
    return validate_invariant_section(contracts, "executor_binding", group_id)


def validate_evidence_assembly_invariants(
    contracts: dict[str, dict],
    group_id: int
) -> list[ValidationIssue]:
    """Validate evidence_assembly rules are identical across group."""
    return validate_invariant_section(
        contracts,
        "evidence_assembly.assembly_rules",
        group_id
    )


def validate_group(group_id: int, verbose: bool = False) -> GroupValidationResult:
    """Validate a single equivalence group."""
    expected_qnums = get_expected_question_numbers_for_group(group_id)

    # Load all contracts in this group
    contracts = {}
    base_slot = None

    for qnum in expected_qnums:
        qid = f"Q{qnum:03d}"
        path = CONTRACTS_DIR / f"{qid}.v3.json"

        if path.exists():
            contract = load_contract(path)
            if contract:
                contracts[qid] = contract
                if not base_slot:
                    base_slot = contract.get("identity", {}).get("base_slot", "UNKNOWN")

    result = GroupValidationResult(
        group_id=group_id,
        base_slot=base_slot or "UNKNOWN",
        contracts_loaded=len(contracts),
        contracts_expected=len(expected_qnums)
```

```
    )

    if len(contracts) < 2:
        result.issues.append(ValidationIssue(
            severity="WARNING",
            group_id=group_id,
            question_ids=list(contracts.keys()),
            component="loading",
                    message=f"Only  {len(contracts)}  contracts  loaded,  cannot  validate
equivalence"
        ))
        return result

    # Run all validations
    result.issues.extend(validate_identity_consistency(contracts, group_id))
    result.issues.extend(validate_executor_binding_invariants(contracts, group_id))
    result.issues.extend(validate_method_binding_invariants(contracts, group_id))
    result.issues.extend(validate_evidence_assembly_invariants(contracts, group_id))
    result.issues.extend(validate_output_contract_consts(contracts, group_id))

    return result


# === Reporting ===

def print_group_result(result: GroupValidationResult, verbose: bool = False) -> None:
    """Print validation result for a group."""
    status = "?" if result.is_valid else "?"
            print(f"\n{status}   Group   {result.group_id:2d}   |   {result.base_slot}   |
{result.contracts_loaded}/{result.contracts_expected} contracts")

    if result.issues:
        for issue in result.issues:
            icon = {"ERROR": "?", "WARNING": "?", "INFO": "?"}.get(issue.severity, "?")
            print(f"   {icon} [{issue.component}] {issue.message}")
            if verbose and issue.details:
                for k, v in issue.details.items():
                    print(f"      ? {k}: {v}")


def print_summary(results: list[GroupValidationResult]) -> None:
    """Print summary of all validation results."""
    total_groups = len(results)
    valid_groups = sum(1 for r in results if r.is_valid)
    total_issues = sum(len(r.issues) for r in results)
    error_count = sum(1 for r in results for i in r.issues if i.severity == "ERROR")
    warning_count = sum(1 for r in results for i in r.issues if i.severity == "WARNING")

    print("\n" + "=" * 60)
    print("VALIDATION SUMMARY")
    print("=" * 60)
    print(f"Groups validated: {total_groups}")
                    print(f"Groups    passing:              {valid_groups}/{total_groups}
({100*valid_groups/total_groups:.1f}%)")
```

```
    print(f"Total issues:     {total_issues}")
    print(f"  ? Errors:      {error_count}")
    print(f"  ? Warnings:    {warning_count}")

    if error_count > 0:
        print("\n??  VALIDATION FAILED - Errors found that require attention")
    else:
        print("\n? VALIDATION PASSED - No critical errors found")


# === Main ===

def main():
    parser = argparse.ArgumentParser(description="Validate executor contract equivalence across groups")
     parser.add_argument("--group", "-g", type=int, help="Validate only specific group (0-29)")
      parser.add_argument("--verbose", "-v", action="store_true", help="Show detailed issue information")
       parser.add_argument("--fix", action="store_true", help="Attempt to auto-fix output_contract const issues")
    args = parser.parse_args()

    print("=" * 60)
    print("EXECUTOR CONTRACT EQUIVALENCE VALIDATOR")
    print("=" * 60)
    print(f"Contracts directory: {CONTRACTS_DIR}")
    print(f"Matrix: {NUM_BASE_SLOTS} base slots × {NUM_POLICY_AREAS} policy areas = 300 contracts")

    if args.group is not None:
        if 0 <= args.group < NUM_BASE_SLOTS:
            groups_to_validate = [args.group]
        else:
            print(f"Error: --group must be 0-{NUM_BASE_SLOTS-1}")
            return 1
    else:
        groups_to_validate = range(NUM_BASE_SLOTS)

    results = []
    for group_id in groups_to_validate:
        result = validate_group(group_id, verbose=args.verbose)
        results.append(result)
        print_group_result(result, verbose=args.verbose)

    print_summary(results)

    return 0 if all(r.is_valid for r in results) else 1


if __name__ == "__main__":
    exit(main())
```

scripts/verify_contract_sync.py

```
#!/usr/bin/env python3
"""
Contract Synchronization Verifier
==================================

Comprehensive verification system that validates contract synchronization results
and ensures deep structural integrity across executor contract groups.

This script performs:
1. Post-sync validation to ensure fixes were correctly applied
2. Cross-contract consistency checking within groups
3. Method-evidence graph validation
4. Human-answer structure completeness verification
5. Golden contract compliance checking
6. Execution path validation

Usage:
    python verify_contract_sync.py --group 0                    # Verify group 0
       python  verify_contract_sync.py  --contracts  Q001,Q031,Q061     #  Verify  specific
contracts
    python verify_contract_sync.py --all --strict               # Verify all with strict
mode
    python verify_contract_sync.py --golden Q001 --compare Q031 # Compare against golden
"""

import argparse
import json
import hashlib
from pathlib import Path
from datetime import datetime, timezone
from typing import Any, Dict, List, Set, Tuple, Optional
from dataclasses import dataclass, field
from enum import Enum
from collections import defaultdict, Counter
import networkx as nx
from termcolor import colored
import sys

# === Configuration ===
CONTRACTS_DIR               =               Path(__file__).parent.parent               /
"src/canonic_phases/Phase_two/json_files_phase_two/executor_contracts/specialized"

# === Verification Rules ===

class VerificationLevel(Enum):
    """Verification strictness levels."""
    CRITICAL = "critical"     # Must pass for execution
    REQUIRED = "required"     # Should pass for correctness
    RECOMMENDED = "recommended"  # Should pass for quality
    OPTIONAL = "optional"     # Nice to have
```

```
class VerificationStatus(Enum):
    """Verification result status."""
    PASSED = "passed"
    FAILED = "failed"
    WARNING = "warning"
    SKIPPED = "skipped"


@dataclass
class VerificationRule:
    """A single verification rule."""
    name: str
    description: str
    level: VerificationLevel
    check_function: str  # Name of function to call
    applies_to: str  # "individual", "group", "all"


@dataclass
class VerificationResult:
    """Result of a single verification check."""
    rule_name: str
    status: VerificationStatus
    message: str
    level: VerificationLevel
    details: Dict[str, Any] = field(default_factory=dict)
    contracts_affected: List[str] = field(default_factory=list)

    def to_dict(self) -> Dict:
        return {
            "rule": self.rule_name,
            "status": self.status.value,
            "level": self.level.value,
            "message": self.message,
            "details": self.details,
            "contracts_affected": self.contracts_affected
        }


@dataclass
class VerificationReport:
    """Complete verification report."""
    timestamp: str
    contracts_verified: List[str]
    golden_contract: Optional[str]
    results: List[VerificationResult] = field(default_factory=list)

    @property
    def passed_count(self) -> int:
        return len([r for r in self.results if r.status == VerificationStatus.PASSED])

    @property
    def failed_count(self) -> int:
        return len([r for r in self.results if r.status == VerificationStatus.FAILED])
```

```
    @property
    def warning_count(self) -> int:
        return len([r for r in self.results if r.status == VerificationStatus.WARNING])

    @property
    def critical_failures(self) -> List[VerificationResult]:
        return [r for r in self.results
                if r.status == VerificationStatus.FAILED
                and r.level == VerificationLevel.CRITICAL]

    def add_result(self, result: VerificationResult):
        self.results.append(result)

    def to_dict(self) -> Dict:
        return {
            "timestamp": self.timestamp,
            "contracts_verified": self.contracts_verified,
            "golden_contract": self.golden_contract,
            "summary": {
                "total_checks": len(self.results),
                "passed": self.passed_count,
                "failed": self.failed_count,
                "warnings": self.warning_count,
                "critical_failures": len(self.critical_failures)
            },
            "results": [r.to_dict() for r in self.results]
        }

    def print_summary(self, verbose: bool = False):
        """Print colored summary to console."""
        print("\n" + "="*80)
        print(colored("VERIFICATION REPORT", "cyan", attrs=["bold"]))
        print("="*80)

        # Summary stats
        print(f"\n? Summary:")
        print(f"  Total checks: {len(self.results)}")
        print(f"  ? Passed: {colored(str(self.passed_count), 'green')}")
        print(f"  ? Failed: {colored(str(self.failed_count), 'red')}")
        print(f"  ??  Warnings: {colored(str(self.warning_count), 'yellow')}")

        if self.critical_failures:
               print(f"\n? {colored(f'CRITICAL FAILURES: {len(self.critical_failures)}',
'red', attrs=['bold'])}")
            for failure in self.critical_failures:
                print(f"  ? {failure.rule_name}: {failure.message}")
                if failure.contracts_affected:
                    print(f"    Affects: {', '.join(failure.contracts_affected)}")

        if verbose or self.failed_count > 0:
            print(f"\n? Detailed Results:")

            # Group by level
```

```
            for level in VerificationLevel:
                level_results = [r for r in self.results if r.level == level]
                if not level_results:
                    continue

                print(f"\n  {level.value.upper()} checks:")
                for result in level_results:
                    if result.status == VerificationStatus.PASSED:
                        symbol = colored("?", "green")
                    elif result.status == VerificationStatus.FAILED:
                        symbol = colored("?", "red")
                    elif result.status == VerificationStatus.WARNING:
                        symbol = colored("!", "yellow")
                    else:
                        symbol = colored("-", "gray")

                    print(f"    {symbol} {result.rule_name}: {result.message}")

                    if verbose and result.details:
                        for key, value in result.details.items():
                            print(f"      ? {key}: {value}")


# === Verification Rules Registry ===

VERIFICATION_RULES = [
    # Critical rules
    VerificationRule(
        name="identity_consistency",
               description="Verify  identity  fields  match  between  identity  block  and
output_contract schema",
        level=VerificationLevel.CRITICAL,
        check_function="verify_identity_consistency",
        applies_to="individual"
    ),
    VerificationRule(
        name="method_evidence_alignment",
        description="Verify all evidence assembly sources map to method provides",
        level=VerificationLevel.CRITICAL,
        check_function="verify_method_evidence_alignment",
        applies_to="individual"
    ),
    VerificationRule(
        name="execution_sequence_validity",
        description="Verify execution sequence has valid dependencies",
        level=VerificationLevel.CRITICAL,
        check_function="verify_execution_sequence",
        applies_to="individual"
    ),

    # Required rules
    VerificationRule(
        name="group_structural_consistency",
        description="Verify shared structures are identical within group",
```

```
        level=VerificationLevel.REQUIRED,
        check_function="verify_group_consistency",
        applies_to="group"
    ),
    VerificationRule(
        name="human_answer_structure_complete",
        description="Verify human_answer_structure has all required components",
        level=VerificationLevel.REQUIRED,
        check_function="verify_human_answer_structure",
        applies_to="individual"
    ),
    VerificationRule(
        name="method_outputs_documented",
        description="Verify all methods have documented outputs",
        level=VerificationLevel.REQUIRED,
        check_function="verify_method_outputs",
        applies_to="individual"
    ),

    # Recommended rules
    VerificationRule(
        name="evidence_nexus_migration",
        description="Verify using EvidenceNexus instead of legacy EvidenceAssembler",
        level=VerificationLevel.RECOMMENDED,
        check_function="verify_evidence_nexus",
        applies_to="individual"
    ),
    VerificationRule(
        name="contract_hash_valid",
        description="Verify contract hash matches content",
        level=VerificationLevel.RECOMMENDED,
        check_function="verify_contract_hash",
        applies_to="individual"
    ),
    VerificationRule(
        name="pattern_coverage",
        description="Verify sufficient pattern coverage for question",
        level=VerificationLevel.RECOMMENDED,
        check_function="verify_pattern_coverage",
        applies_to="individual"
    )
]


# === Helper Functions ===

def deep_get(obj: Dict, path: str) -> Any:
    """Get nested dict value using dot notation."""
    keys = path.split('.')
    current = obj
    for key in keys:
        if isinstance(current, dict):
            current = current.get(key)
            if current is None:
```

```
                return None
            elif isinstance(current, list) and key.isdigit():
                idx = int(key)
                if 0 <= idx < len(current):
                    current = current[idx]
                else:
                    return None
            else:
                return None
        return current


def load_contract(question_id: str) -> Optional[Dict]:
    """Load contract JSON."""
    path = CONTRACTS_DIR / f"{question_id}.v3.json"
    if not path.exists():
        return None
    try:
        with open(path, 'r', encoding='utf-8') as f:
            return json.load(f)
    except Exception as e:
        print(f"Error loading {question_id}: {e}")
        return None


def compute_contract_hash(contract: Dict) -> str:
    """Compute SHA-256 hash of contract content."""
    contract_copy = json.loads(json.dumps(contract))
    if "identity" in contract_copy:
        contract_copy["identity"].pop("contract_hash", None)
        contract_copy["identity"].pop("created_at", None)
        contract_copy["identity"].pop("updated_at", None)
    content = json.dumps(contract_copy, sort_keys=True, ensure_ascii=False)
    return hashlib.sha256(content.encode()).hexdigest()


# === Individual Contract Verification Functions ===

def verify_identity_consistency(contract: Dict, qid: str) -> VerificationResult:
        """Verify identity fields match between identity block and output_contract
schema."""

    fields_to_check = [
        "base_slot",
        "question_id",
        "question_global",
        "policy_area_id",
        "dimension_id",
        "cluster_id"
    ]

    mismatches = []

    for field in fields_to_check:
```

```
        identity_value = deep_get(contract, f"identity.{field}")
                                                schema_value      =      deep_get(contract,
f"output_contract.schema.properties.{field}.const")

        # Skip if schema doesn't have this field
        if deep_get(contract, f"output_contract.schema.properties.{field}") is None:
            continue

        if schema_value != identity_value:
            mismatches.append({
                "field": field,
                "identity": identity_value,
                "schema": schema_value
            })

    if mismatches:
        return VerificationResult(
            rule_name="identity_consistency",
            status=VerificationStatus.FAILED,
            level=VerificationLevel.CRITICAL,
            message=f"{len(mismatches)} identity fields don't match schema",
            details={"mismatches": mismatches},
            contracts_affected=[qid]
        )

    return VerificationResult(
        rule_name="identity_consistency",
        status=VerificationStatus.PASSED,
        level=VerificationLevel.CRITICAL,
        message="All identity fields match schema",
        contracts_affected=[qid]
    )


def verify_method_evidence_alignment(contract: Dict, qid: str) -> VerificationResult:
    """Verify all evidence assembly sources map to method provides."""

    # Get all method provides
    methods = deep_get(contract, "method_binding.methods") or []
    provides_set = {m.get("provides") for m in methods if m.get("provides")}

    # Get all referenced sources in assembly rules
    assembly_rules = deep_get(contract, "evidence_assembly.assembly_rules") or []
    referenced_sources = set()
    unmapped_sources = []

    for rule in assembly_rules:
        sources = rule.get("sources", [])
        for source in sources:
            # Handle wildcards
            if "*" in source:
                base = source.split("*")[0].rstrip(".")
                if base:
                    referenced_sources.add(base)
```

```
                    # Check if any provides starts with this base
                    if not any(p.startswith(base) for p in provides_set):
                        unmapped_sources.append(source)
            else:
                referenced_sources.add(source)
                if source not in provides_set:
                    unmapped_sources.append(source)

    if unmapped_sources:
        return VerificationResult(
            rule_name="method_evidence_alignment",
            status=VerificationStatus.FAILED,
            level=VerificationLevel.CRITICAL,
            message=f"{len(unmapped_sources)} assembly sources don't map to methods",
            details={
                "unmapped_sources": unmapped_sources,
                "available_provides": sorted(list(provides_set))
            },
            contracts_affected=[qid]
        )

    # Calculate coverage
    coverage = len(referenced_sources) / len(provides_set) if provides_set else 0

    if coverage < 0.5:
        return VerificationResult(
            rule_name="method_evidence_alignment",
            status=VerificationStatus.WARNING,
            level=VerificationLevel.CRITICAL,
            message=f"Only {coverage:.0%} of methods are used in assembly",
            details={"coverage": coverage},
            contracts_affected=[qid]
        )

    return VerificationResult(
        rule_name="method_evidence_alignment",
        status=VerificationStatus.PASSED,
        level=VerificationLevel.CRITICAL,
        message=f"All assembly sources map to methods ({coverage:.0%} coverage)",
        contracts_affected=[qid]
    )


def verify_execution_sequence(contract: Dict, qid: str) -> VerificationResult:
    """Verify execution sequence forms a valid DAG."""

    exec_seq = deep_get(contract, "method_binding.execution_sequence")

    if not exec_seq:
        return VerificationResult(
            rule_name="execution_sequence_validity",
            status=VerificationStatus.WARNING,
            level=VerificationLevel.CRITICAL,
            message="No execution_sequence defined",
```

```
            contracts_affected=[qid]
        )

    if not isinstance(exec_seq, dict):
        return VerificationResult(
            rule_name="execution_sequence_validity",
            status=VerificationStatus.FAILED,
            level=VerificationLevel.CRITICAL,
            message="execution_sequence is not a dictionary",
            contracts_affected=[qid]
        )

    # Build dependency graph
    G = nx.DiGraph()

    for step_name, step_info in exec_seq.items():
        G.add_node(step_name)

        deps = step_info.get("depends_on", [])
        if isinstance(deps, list):
            for dep in deps:
                G.add_edge(dep, step_name)

    # Check for cycles
    if not nx.is_directed_acyclic_graph(G):
        cycles = list(nx.simple_cycles(G))
        return VerificationResult(
            rule_name="execution_sequence_validity",
            status=VerificationStatus.FAILED,
            level=VerificationLevel.CRITICAL,
            message=f"Execution sequence has circular dependencies",
            details={"cycles": cycles},
            contracts_affected=[qid]
        )

    # Check all dependencies exist
    all_steps = set(exec_seq.keys())
    missing_deps = []

    for step_name, step_info in exec_seq.items():
        deps = step_info.get("depends_on", [])
        if isinstance(deps, list):
            for dep in deps:
                if dep not in all_steps:
                    missing_deps.append(f"{step_name} depends on missing {dep}")

    if missing_deps:
        return VerificationResult(
            rule_name="execution_sequence_validity",
            status=VerificationStatus.FAILED,
            level=VerificationLevel.CRITICAL,
            message=f"Execution sequence has missing dependencies",
            details={"missing_dependencies": missing_deps},
            contracts_affected=[qid]
```

```
        )

    return VerificationResult(
        rule_name="execution_sequence_validity",
        status=VerificationStatus.PASSED,
        level=VerificationLevel.CRITICAL,
        message=f"Valid DAG with {len(G.nodes)} steps",
        details={"num_steps": len(G.nodes), "num_edges": len(G.edges)},
        contracts_affected=[qid]
    )


def verify_human_answer_structure(contract: Dict, qid: str) -> VerificationResult:
    """Verify human_answer_structure completeness."""

    has = contract.get("human_answer_structure", {})

    if not has:
        return VerificationResult(
            rule_name="human_answer_structure_complete",
            status=VerificationStatus.FAILED,
            level=VerificationLevel.REQUIRED,
            message="Missing human_answer_structure",
            contracts_affected=[qid]
        )

    required_components = [
        "evidence_structure_schema",
        "concrete_example",
        "validation_against_expected_elements",
        "assembly_flow"
    ]

    missing = []
    for component in required_components:
        if component not in has:
            missing.append(component)

    if missing:
        return VerificationResult(
            rule_name="human_answer_structure_complete",
            status=VerificationStatus.FAILED,
            level=VerificationLevel.REQUIRED,
            message=f"Missing {len(missing)} required components",
            details={"missing_components": missing},
            contracts_affected=[qid]
        )

    # Check schema validity
    schema = has.get("evidence_structure_schema", {})
    if not schema.get("properties"):
        return VerificationResult(
            rule_name="human_answer_structure_complete",
            status=VerificationStatus.WARNING,
```

```
            level=VerificationLevel.REQUIRED,
            message="Evidence schema has no properties defined",
            contracts_affected=[qid]
        )

    # Check example validity
    example = has.get("concrete_example", {})
    if not example:
        return VerificationResult(
            rule_name="human_answer_structure_complete",
            status=VerificationStatus.WARNING,
            level=VerificationLevel.REQUIRED,
            message="Concrete example is empty",
            contracts_affected=[qid]
        )

    return VerificationResult(
        rule_name="human_answer_structure_complete",
        status=VerificationStatus.PASSED,
        level=VerificationLevel.REQUIRED,
        message="Complete human_answer_structure with all components",
        details={
            "schema_properties": len(schema.get("properties", {})),
            "example_fields": len(example)
        },
        contracts_affected=[qid]
    )


def verify_method_outputs(contract: Dict, qid: str) -> VerificationResult:
    """Verify all methods have documented outputs."""

    methods = deep_get(contract, "method_binding.methods") or []
    method_outputs = contract.get("method_outputs", {})

    if not method_outputs:
        return VerificationResult(
            rule_name="method_outputs_documented",
            status=VerificationStatus.FAILED,
            level=VerificationLevel.REQUIRED,
            message="No method_outputs section",
            contracts_affected=[qid]
        )

    undocumented = []
    incomplete = []

    for method in methods:
        method_name = method.get("method_name")
        class_name = method.get("class_name")
        full_name = f"{class_name}.{method_name}"

        if full_name not in method_outputs:
            undocumented.append(full_name)
```

```
        else:
            output_doc = method_outputs[full_name]
            # Check for required fields
                        required_fields  =  ["output_type",  "structure",  "validation",
"usage_in_assembly"]
            missing_fields = [f for f in required_fields if f not in output_doc]
            if missing_fields:
                incomplete.append(f"{full_name}: missing {missing_fields}")

    if undocumented:
        return VerificationResult(
            rule_name="method_outputs_documented",
            status=VerificationStatus.FAILED,
            level=VerificationLevel.REQUIRED,
            message=f"{len(undocumented)} methods lack output documentation",
            details={"undocumented_methods": undocumented},
            contracts_affected=[qid]
        )

    if incomplete:
        return VerificationResult(
            rule_name="method_outputs_documented",
            status=VerificationStatus.WARNING,
            level=VerificationLevel.REQUIRED,
            message=f"{len(incomplete)} methods have incomplete documentation",
            details={"incomplete_methods": incomplete},
            contracts_affected=[qid]
        )

    return VerificationResult(
        rule_name="method_outputs_documented",
        status=VerificationStatus.PASSED,
        level=VerificationLevel.REQUIRED,
        message=f"All {len(methods)} methods fully documented",
        contracts_affected=[qid]
    )


def verify_evidence_nexus(contract: Dict, qid: str) -> VerificationResult:
    """Verify using EvidenceNexus instead of legacy EvidenceAssembler."""

    class_name = deep_get(contract, "evidence_assembly.class_name")
    module_name = deep_get(contract, "evidence_assembly.module")

    if class_name == "EvidenceAssembler":
        return VerificationResult(
            rule_name="evidence_nexus_migration",
            status=VerificationStatus.WARNING,
            level=VerificationLevel.RECOMMENDED,
            message="Still using legacy EvidenceAssembler",
            details={
                "current_class": class_name,
                "current_module": module_name,
                "recommended_class": "EvidenceNexus",
```

```
                "recommended_module": "farfan_core.core.orchestrator.evidence_nexus"
            },
            contracts_affected=[qid]
        )

    if class_name != "EvidenceNexus":
        return VerificationResult(
            rule_name="evidence_nexus_migration",
            status=VerificationStatus.WARNING,
            level=VerificationLevel.RECOMMENDED,
            message=f"Using unknown evidence class: {class_name}",
            contracts_affected=[qid]
        )

    # Check for evidence_structure_post_nexus
    if not deep_get(contract, "evidence_structure_post_nexus"):
        return VerificationResult(
            rule_name="evidence_nexus_migration",
            status=VerificationStatus.WARNING,
            level=VerificationLevel.RECOMMENDED,
            message="Using EvidenceNexus but missing evidence_structure_post_nexus",
            contracts_affected=[qid]
        )

    return VerificationResult(
        rule_name="evidence_nexus_migration",
        status=VerificationStatus.PASSED,
        level=VerificationLevel.RECOMMENDED,
        message="Properly using EvidenceNexus with post-nexus structure",
        contracts_affected=[qid]
    )


def verify_contract_hash(contract: Dict, qid: str) -> VerificationResult:
    """Verify contract hash matches content."""

    stored_hash = deep_get(contract, "identity.contract_hash")

    if not stored_hash:
        return VerificationResult(
            rule_name="contract_hash_valid",
            status=VerificationStatus.WARNING,
            level=VerificationLevel.RECOMMENDED,
            message="No contract hash in identity",
            contracts_affected=[qid]
        )

    computed_hash = compute_contract_hash(contract)

    if stored_hash != computed_hash:
        return VerificationResult(
            rule_name="contract_hash_valid",
            status=VerificationStatus.WARNING,
            level=VerificationLevel.RECOMMENDED,
```

```
            message="Contract hash doesn't match content",
            details={
                "stored": stored_hash[:16] + "...",
                "computed": computed_hash[:16] + "..."
            },
            contracts_affected=[qid]
        )

    return VerificationResult(
        rule_name="contract_hash_valid",
        status=VerificationStatus.PASSED,
        level=VerificationLevel.RECOMMENDED,
        message="Contract hash valid",
        contracts_affected=[qid]
    )


def verify_pattern_coverage(contract: Dict, qid: str) -> VerificationResult:
    """Verify sufficient pattern coverage for question."""

    patterns = deep_get(contract, "question_context.patterns") or []

    if len(patterns) < 5:
        return VerificationResult(
            rule_name="pattern_coverage",
            status=VerificationStatus.WARNING,
            level=VerificationLevel.RECOMMENDED,
            message=f"Only {len(patterns)} patterns defined (recommended: 10+)",
            contracts_affected=[qid]
        )

    # Check pattern diversity (categories)
    categories = set()
    for pattern in patterns:
        if isinstance(pattern, dict):
            cat = pattern.get("category", "GENERAL")
            categories.add(cat)

    if len(categories) < 3:
        return VerificationResult(
            rule_name="pattern_coverage",
            status=VerificationStatus.WARNING,
            level=VerificationLevel.RECOMMENDED,
            message=f"Low pattern diversity: only {len(categories)} categories",
            details={"categories": list(categories)},
            contracts_affected=[qid]
        )

    return VerificationResult(
        rule_name="pattern_coverage",
        status=VerificationStatus.PASSED,
        level=VerificationLevel.RECOMMENDED,
           message=f"Good pattern coverage: {len(patterns)} patterns, {len(categories)}
categories",
```

```
        contracts_affected=[qid]
    )


# === Group Verification Functions ===

def verify_group_consistency(contracts: Dict[str, Dict]) -> VerificationResult:
    """Verify shared structures are identical within group."""

    if len(contracts) < 2:
        return VerificationResult(
            rule_name="group_structural_consistency",
            status=VerificationStatus.SKIPPED,
            level=VerificationLevel.REQUIRED,
            message="Need at least 2 contracts for group verification"
        )

    shared_fields = [
        "executor_binding",
        "method_binding.methods",
        "method_binding.orchestration_mode",
        "evidence_assembly.module",
        "evidence_assembly.class_name",
    ]

    inconsistencies = []

    for field_path in shared_fields:
        values = {}
        for qid, contract in contracts.items():
            value = deep_get(contract, field_path)
            if value is not None:
                # Convert to string for comparison
                value_str = json.dumps(value, sort_keys=True)
                if value_str not in values:
                    values[value_str] = []
                values[value_str].append(qid)

        if len(values) > 1:
            inconsistencies.append({
                "field": field_path,
                "variants": len(values),
                "distribution": {
                    f"variant_{i}": qids
                    for i, qids in enumerate(values.values())
                }
            })

    if inconsistencies:
        return VerificationResult(
            rule_name="group_structural_consistency",
            status=VerificationStatus.FAILED,
            level=VerificationLevel.REQUIRED,
            message=f"{len(inconsistencies)} shared fields are inconsistent",
```

```
            details={"inconsistencies": inconsistencies},
            contracts_affected=list(contracts.keys())
        )

    return VerificationResult(
        rule_name="group_structural_consistency",
        status=VerificationStatus.PASSED,
        level=VerificationLevel.REQUIRED,
        message="All shared structures are consistent",
        contracts_affected=list(contracts.keys())
    )


# === Verification Runner ===

class ContractVerifier:
    """Main verification engine."""

    def __init__(self, strict_mode: bool = False):
        self.strict_mode = strict_mode

    def verify_contracts(
        self,
        contracts: Dict[str, Dict],
        golden_id: Optional[str] = None
    ) -> VerificationReport:
        """Run all applicable verification rules."""

        report = VerificationReport(
            timestamp=datetime.now(timezone.utc).isoformat(),
            contracts_verified=list(contracts.keys()),
            golden_contract=golden_id
        )

        # Run individual contract checks
        for qid, contract in contracts.items():
            for rule in VERIFICATION_RULES:
                if rule.applies_to != "individual":
                    continue

                # Get verification function
                func_name = rule.check_function
                if hasattr(sys.modules[__name__], func_name):
                    func = getattr(sys.modules[__name__], func_name)
                    try:
                        result = func(contract, qid)
                        report.add_result(result)
                    except Exception as e:
                        report.add_result(VerificationResult(
                            rule_name=rule.name,
                            status=VerificationStatus.FAILED,
                            level=rule.level,
                            message=f"Verification failed with error: {str(e)}",
                            contracts_affected=[qid]
```

```
                ))

    # Run group checks
    for rule in VERIFICATION_RULES:
        if rule.applies_to != "group":
            continue

        func_name = rule.check_function
        if hasattr(sys.modules[__name__], func_name):
            func = getattr(sys.modules[__name__], func_name)
            try:
                result = func(contracts)
                report.add_result(result)
            except Exception as e:
                report.add_result(VerificationResult(
                    rule_name=rule.name,
                    status=VerificationStatus.FAILED,
                    level=rule.level,
                    message=f"Verification failed with error: {str(e)}",
                    contracts_affected=list(contracts.keys())
                ))

    # Golden contract comparison if specified
    if golden_id and golden_id in contracts:
        self._verify_against_golden(contracts, golden_id, report)

    return report

def _verify_against_golden(
    self,
    contracts: Dict[str, Dict],
    golden_id: str,
    report: VerificationReport
):
    """Additional checks comparing against golden contract."""

    golden = contracts[golden_id]

    for qid, contract in contracts.items():
        if qid == golden_id:
            continue

        # Check critical structures exist if in golden
        critical_in_golden = [
            "method_binding.execution_sequence",
            "method_outputs",
            "evidence_structure_post_nexus",
            "human_answer_structure"
        ]

        for path in critical_in_golden:
            if deep_get(golden, path) and not deep_get(contract, path):
                report.add_result(VerificationResult(
                    rule_name="golden_compliance",
```

```
                        status=VerificationStatus.WARNING,
                        level=VerificationLevel.RECOMMENDED,
                        message=f"Missing structure present in golden: {path}",
                        contracts_affected=[qid]
                    ))


# === Report Generation ===

def generate_html_report(report: VerificationReport, output_path: Path):
    """Generate an HTML verification report."""

    # Status colors
    status_colors = {
        VerificationStatus.PASSED: "#27ae60",
        VerificationStatus.FAILED: "#e74c3c",
        VerificationStatus.WARNING: "#f39c12",
        VerificationStatus.SKIPPED: "#95a5a6"
    }

    # Level badges
    level_badges = {
        VerificationLevel.CRITICAL: "?",
        VerificationLevel.REQUIRED: "??",
        VerificationLevel.RECOMMENDED: "?",
        VerificationLevel.OPTIONAL: "??"
    }

    html = f"""
<!DOCTYPE html>
<html>
<head>
    <title>Contract Verification Report</title>
    <style>
        body {{
             font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, Oxygen,
Ubuntu, sans-serif;
            margin: 0;
            padding: 20px;
            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
            min-height: 100vh;
        }}
        .container {{
            max-width: 1200px;
            margin: auto;
            background: white;
            border-radius: 16px;
            box-shadow: 0 20px 60px rgba(0,0,0,0.3);
            overflow: hidden;
        }}
        .header {{
            background: linear-gradient(135deg, #2c3e50 0%, #34495e 100%);
            color: white;
            padding: 40px;
```

```
}}
h1 {{
    margin: 0;
    font-size: 2.5em;
    font-weight: 300;
}}
.timestamp {{
    opacity: 0.9;
    margin-top: 10px;
}}
.summary-grid {{
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
    gap: 20px;
    padding: 40px;
    background: #f8f9fa;
}}
.stat-card {{
    background: white;
    padding: 25px;
    border-radius: 12px;
    box-shadow: 0 2px 10px rgba(0,0,0,0.08);
    text-align: center;
    transition: transform 0.2s;
}}
.stat-card:hover {{
    transform: translateY(-5px);
}}
.stat-value {{
    font-size: 3em;
    font-weight: bold;
    margin-bottom: 10px;
}}
.stat-label {{
    color: #7f8c8d;
    text-transform: uppercase;
    font-size: 0.85em;
    letter-spacing: 1px;
}}
.results {{
    padding: 40px;
}}
.result-group {{
    margin-bottom: 30px;
}}
.result-header {{
    font-size: 1.3em;
    font-weight: 600;
    color: #2c3e50;
    margin-bottom: 15px;
    padding-bottom: 10px;
    border-bottom: 2px solid #ecf0f1;
}}
.result-item {{
```

```
    background: white;
    border-left: 4px solid #95a5a6;
    padding: 15px 20px;
    margin: 10px 0;
    border-radius: 4px;
    transition: all 0.2s;
}}
.result-item:hover {{
    box-shadow: 0 2px 8px rgba(0,0,0,0.1);
}}
.result-item.passed {{ border-left-color: #27ae60; background: #f0fdf4; }}
.result-item.failed {{ border-left-color: #e74c3c; background: #fef2f2; }}
.result-item.warning {{ border-left-color: #f39c12; background: #fffbeb; }}
.result-item.skipped {{ border-left-color: #95a5a6; background: #f8f9fa; }}
.result-title {{
    display: flex;
    justify-content: space-between;
    align-items: center;
    margin-bottom: 5px;
}}
.result-name {{
    font-weight: 600;
    color: #2c3e50;
}}
.result-status {{
    padding: 3px 10px;
    border-radius: 12px;
    font-size: 0.85em;
    font-weight: 600;
    text-transform: uppercase;
}}
.result-message {{
    color: #5a6c7d;
    margin: 5px 0;
}}
.result-details {{
    margin-top: 10px;
    padding: 10px;
    background: rgba(0,0,0,0.03);
    border-radius: 4px;
    font-family: 'Courier New', monospace;
    font-size: 0.9em;
    color: #34495e;
}}
.critical-section {{
    background: #fef2f2;
    border: 2px solid #e74c3c;
    border-radius: 8px;
    padding: 20px;
    margin: 20px 40px;
}}
.critical-title {{
    color: #e74c3c;
    font-size: 1.2em;
```

```
            font-weight: bold;
            margin-bottom: 15px;
        }}
        .footer {{
            text-align: center;
            padding: 30px;
            background: #f8f9fa;
            color: #7f8c8d;
            border-top: 1px solid #e0e6ed;
        }}
    </style>
</head>
<body>
    <div class="container">
        <div class="header">
            <h1>? Contract Verification Report</h1>
            <div class="timestamp">Generated: {report.timestamp}</div>
                                                <div    class="timestamp">Contracts:    {',
'.join(report.contracts_verified[:5])}{'...' if len(report.contracts_verified) > 5 else
''}</div>
        </div>

        <div class="summary-grid">
            <div class="stat-card">
                                                <div   class="stat-value"   style="color:
#27ae60;">{report.passed_count}</div>
                <div class="stat-label">Passed</div>
            </div>
            <div class="stat-card">
                                                <div   class="stat-value"   style="color:
#e74c3c;">{report.failed_count}</div>
                <div class="stat-label">Failed</div>
            </div>
            <div class="stat-card">
                                                <div   class="stat-value"   style="color:
#f39c12;">{report.warning_count}</div>
                <div class="stat-label">Warnings</div>
            </div>
            <div class="stat-card">
                                                <div   class="stat-value"   style="color:
#3498db;">{len(report.results)}</div>
                <div class="stat-label">Total Checks</div>
            </div>
        </div>
"""

    # Critical failures section
    if report.critical_failures:
        html += f"""
        <div class="critical-section">
            <div class="critical-title">? Critical Failures Detected</div>
"""
        for failure in report.critical_failures:
            html += f"""
```

```
            <div style="margin: 10px 0;">
                <strong>{failure.rule_name}</strong>: {failure.message}
                        {f'<div style="margin-left: 20px; color: #7f8c8d;">Affects: {",
".join(failure.contracts_affected)}</div>' if failure.contracts_affected else ''}
            </div>
"""
        html += "</div>"

    # Results by level
    html += '<div class="results">'

    for level in VerificationLevel:
        level_results = [r for r in report.results if r.level == level]
        if not level_results:
            continue

        html += f"""
        <div class="result-group">
            <div class="result-header">
                {level_badges.get(level, '')} {level.value.title()} Checks
            </div>
"""

        for result in level_results:
            status_color = status_colors.get(result.status, "#95a5a6")
            html += f"""
            <div class="result-item {result.status.value}">
                <div class="result-title">
                            <span class="result-name">{result.rule_name.replace('_', '
').title()}</span>
                        <span class="result-status" style="background: {status_color};
color: white;">
                            {result.status.value}
                    </span>
                </div>
                <div class="result-message">{result.message}</div>
"""
            if result.details:
                details_str = json.dumps(result.details, indent=2)
                if len(details_str) > 500:
                    details_str = details_str[:500] + "..."
                html += f'<div class="result-details">{details_str}</div>'

            html += '</div>'

        html += '</div>'

    html += '</div>'

    # Footer
    html += f"""
        <div class="footer">
            <div>Contract Verification System v1.0</div>
            <div style="margin-top: 10px;">
```

```
                {'? All critical checks passed' if not report.critical_failures else f'?
{len(report.critical_failures)} critical failures detected'}
            </div>
        </div>
    </div>
</body>
</html>
"""

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


# === Main ===

def main():
    parser = argparse.ArgumentParser(description="Contract synchronization verifier")
     parser.add_argument("--contracts", type=str, help="Comma-separated question IDs to
verify")
    parser.add_argument("--group", type=int, help="Verify entire group (0-29)")
    parser.add_argument("--all", action="store_true", help="Verify all contracts")
    parser.add_argument("--golden", type=str, help="Golden contract for comparison")
      parser.add_argument("--strict", action="store_true", help="Strict mode - fail on
warnings")
    parser.add_argument("--json-report", type=str, help="Save JSON report")
    parser.add_argument("--html-report", type=str, help="Save HTML report")
    parser.add_argument("--verbose", "-v", action="store_true", help="Verbose output")

    args = parser.parse_args()

    print("="*80)
    print(colored("CONTRACT VERIFICATION SYSTEM", "cyan", attrs=["bold"]))
    print("="*80)

    # Determine contracts to verify
    if args.contracts:
        qids = [q.strip() for q in args.contracts.split(',')]
    elif args.group is not None:
        base = args.group + 1
        qids = [f"Q{base + (i * 30):03d}" for i in range(10)]
    elif args.all:
        qids = [f"Q{i:03d}" for i in range(1, 301)]
    else:
        # Default: first group
        qids = [f"Q{i:03d}" for i in [1, 31, 61, 91, 121, 151, 181, 211, 241, 271]]

    print(f"\n? Loading {len(qids)} contracts...")

    # Load contracts
    contracts = {}
    for qid in qids:
        contract = load_contract(qid)
        if contract:
            contracts[qid] = contract
```

```
            print(f"  ? Loaded {qid}")
        else:
            print(f"  ? Failed to load {qid}")

    if not contracts:
        print("\n? No contracts loaded!")
        return 1

    print(f"\n? Running verification on {len(contracts)} contracts...")

    # Run verification
    verifier = ContractVerifier(strict_mode=args.strict)
    report = verifier.verify_contracts(contracts, golden_id=args.golden)

    # Display results
    report.print_summary(verbose=args.verbose)

    # Save reports
    if args.json_report:
        with open(args.json_report, 'w') as f:
            json.dump(report.to_dict(), f, indent=2)
        print(f"\n? JSON report saved: {args.json_report}")

    if args.html_report:
        generate_html_report(report, Path(args.html_report))
        print(f"? HTML report saved: {args.html_report}")

    # Exit code
    if report.critical_failures:
              print(f"\n?  Verification  failed:  {len(report.critical_failures)}  critical
issues")
        return 1
    elif args.strict and report.failed_count > 0:
        print(f"\n? Strict mode: {report.failed_count} failures")
        return 1
    else:
         print(f"\n? Verification {'passed' if report.failed_count == 0 else 'completed
with warnings'}")
        return 0


if __name__ == "__main__":
    try:
        sys.exit(main())
    except KeyboardInterrupt:
        print("\n\nInterrupted by user")
        sys.exit(1)
    except Exception as e:
        print(f"\n? Fatal error: {e}")
        import traceback
        traceback.print_exc()
        sys.exit(1)
```

```
scripts/verify_dashboard_wiring.py

import sys
from pathlib import Path
import asyncio

# Setup path
ROOT = Path(__file__).resolve().parent.parent
sys.path.insert(0, str(ROOT))

from src.dashboard_atroz_.ingestion import DashboardIngester

async def main():
    print("Verifying Dashboard Wiring and Data Availability...")
    ingester = DashboardIngester()

    # 1. Populate Ref Data
    try:
        ingester.populate_reference_data()
        print("? Reference data population triggered (Simulated DB).")
    except Exception as e:
        print(f"? Reference data population failed: {e}")

    # 2. Test Ingestion with Fuzzy Name Matching
    print("\nTesting Matching Logic:")

    # Mock Context with filename-based identification
    class MockInputData:
        document_id = "DOC-UNKNOWN-ID"
        # "Argelia" should match Argelia (Cauca) 19050
        pdf_path = "data/Plan_De_Desarrollo_Argelia_Cauca_2024.pdf"

    class MockDoc:
        input_data = MockInputData()

    class MockMacro:
        macro_score = 85.5
        class Details:
            quality_band = "OPTIMAL"
        details = Details()

    class MockCluster:
        cluster_id = "CL01"
        score = 90.0

    context = {
        "document": MockDoc(),
        "macro_result": MockMacro(),
        "cluster_scores": [MockCluster()],
        "scored_results": []
    }

    success = await ingester.ingest_results(context)
    if success:
```

```
        print("? Fuzzy matching & Ingestion successful.")
    else:
        print("? Fuzzy matching failed.")

if __name__ == "__main__":
    asyncio.run(main())
```

```
setup.py

"""F.A.R.F.A.N Setup Configuration"""

from setuptools import setup, find_packages
from pathlib import Path

# Read README for long description
readme_file = Path(__file__).parent / "README.ES.md"
long_description = readme_file.read_text(encoding="utf-8") if readme_file.exists() else
""

setup(
    name="farfan-pipeline",
    version="1.0.0",
       description="Framework for Advanced Retrieval of Administrative Narratives -
Mechanistic Policy Pipeline",
    long_description=long_description,
    long_description_content_type="text/markdown",
    author="F.A.R.F.A.N Development Team",
    python_requires=">=3.12,<3.13",
    packages=find_packages(where="src"),
    package_dir={"": "src"},
    install_requires=[
        # Core API and validation
        "fastapi>=0.109.0",
        "pydantic>=2.0.0",
        "uvicorn>=0.27.0",

        # NLP and transformers
        "transformers>=4.41.0,<4.42.0",
        "sentence-transformers>=3.1.0,<3.2.0",
        "accelerate>=1.2.0",
        "tokenizers>=0.15.0",

        # Bayesian analysis
        "pymc>=5.16.0,<5.17.0",
        "pytensor>=2.25.1,<2.26",
        "arviz>=0.17.0",

        # Machine learning
        "scikit-learn>=1.6.0",
        "numpy>=1.26.4,<2.0.0",
        "scipy>=1.11.0",

        # Graph analysis
        "networkx>=3.0",

        # NLP tools
        "spacy>=3.7.0",

        # PDF processing
        "PyPDF2>=3.0.0",
        "pdfplumber>=0.10.0",
```

```
        # Data handling
        "pandas>=2.1.0",
        "pyarrow>=14.0.0",

        # Configuration
        "python-dotenv>=1.0.0",

        # Utilities
        "requests>=2.31.0",
        "aiohttp>=3.9.0",
    ],
    extras_require={
        "dev": [
            "pytest>=8.0.0",
            "pytest-cov>=4.1.0",
            "pytest-asyncio>=0.23.0",
            "ruff>=0.1.0",
            "mypy>=1.8.0",
            "black>=24.0.0",
        ],
    },
    entry_points={
        "console_scripts": [
            "farfan-pipeline=farfan_pipeline.entrypoint.main:main",
        ],
    },
    classifiers=[
        "Development Status :: 5 - Production/Stable",
        "Intended Audience :: Science/Research",
        "License :: OSI Approved :: MIT License",
        "Programming Language :: Python :: 3.12",
        "Topic :: Scientific/Engineering :: Artificial Intelligence",
    ],
)
```

src/__init__.py

```
"""
F.A.R.F.A.N Pipeline - Source Root Package
==========================================

Root package for the F.A.R.F.A.N Mechanistic Policy Pipeline.

This package contains the core pipeline implementation including:
- Canonic phases (Phase 0-9)
- Cross-cutting infrastructure (SISAS, CAPAZ, Dura Lex)
- Orchestration and wiring
- Dashboard and visualization
- Methods dispensary
"""

__version__ = "CPP-2025.1"
```

src/batch_concurrence/__init__.py

```
"""
Concurrency module for deterministic parallel execution.

This module provides a deterministic WorkerPool for parallel task execution
with controlled max_workers, backoff, abortability, and per-task instrumentation.
"""

from farfan_pipeline.concurrency.concurrency import (
    TaskExecutionError,
    TaskMetrics,
    TaskResult,
    TaskStatus,
    WorkerPool,
    WorkerPoolConfig,
)

__all__ = [
    "WorkerPool",
    "TaskResult",
    "WorkerPoolConfig",
    "TaskExecutionError",
    "TaskStatus",
    "TaskMetrics",
]
```

src/batch_concurrence/concurrency.py

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

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

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

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

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

from __future__ import annotations

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

if TYPE_CHECKING:
    from collections.abc import Callable

logger = logging.getLogger(__name__)

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

class TaskExecutionError(Exception):
```

```
    """Exception raised when task execution fails."""
    pass

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

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

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

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

@dataclass
class TaskResult:
    """Result of a task execution.

    Attributes:
        task_id: Unique task identifier
        task_name: Human-readable task name
```

```
        success: Whether task succeeded
        result: Task result data (None if failed)
        error: Exception if task failed (None if succeeded)
        metrics: Execution metrics
    """
    task_id: str
    task_name: str
    success: bool
    result: Any = None
    error: Exception | None = None
    metrics: TaskMetrics | None = None

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

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

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

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

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

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

```
    )

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

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

    Args:
        retry_count: Number of retries already attempted

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

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

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

    Returns:
        Task result

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

    # Initialize metrics
    with self._lock:
```

```
            self._metrics[task_id] = TaskMetrics(
                task_id=task_id,
                task_name=task_name,
                status=TaskStatus.RUNNING,
                start_time=time.time(),
                worker_id=worker_id
            )

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

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

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

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

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

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

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

                return result

            except Exception as e:
                last_error = e
```

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

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

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

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

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

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

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

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

    def submit_task(
        self,
        task_name: str,
        task_fn: Callable,
        args: tuple = (),
        kwargs: dict[str, Any] | None = None,
    ) -> str:
        """
```

```
    Submit a task for execution.

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

    Returns:
        Unique task identifier

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

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

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

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

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

    return task_id

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

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

    Returns:
        TaskResult with execution metrics

    Raises:
        KeyError: If task_id is not found
```

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

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

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

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

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

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

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

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

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

def wait_for_all(
    self,
    timeout: float | None = None,
```

```
        return_when: str = "ALL_COMPLETED"
    ) -> list[TaskResult]:
        """
        Wait for all submitted tasks to complete.

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

        Returns:
            List of TaskResults for all tasks

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

        start_time = time.time()
        results = []

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

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

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

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

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

                    # Check if we should return early on first exception
```

```
                    if return_when == "FIRST_EXCEPTION" and not result.success:
                        if self.config.enable_instrumentation:
                            logger.warning(
                                                f"Returning early due to task failure:
{result.task_name}"
                            )
                        break

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

            return results

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

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

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

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

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

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

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

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

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

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

        return pending_count

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

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

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

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

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

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

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

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

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

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

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

    if cancel_futures:
        self.abort_pending_tasks()

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

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

    self._is_shutdown = True

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

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

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

src/canonic_phases/__init__.py

```
"""Compatibility shim for legacy imports.

The canonical phases implementation moved to `farfan_pipeline.phases`.
This module preserves historical imports like:

- `import canonic_phases.Phase_zero...`

New code should prefer:
- `import farfan_pipeline.phases...`
"""

from __future__ import annotations

from pathlib import Path


# Redirect package submodule resolution to the new location.
__path__ = [
               str((Path(__file__).resolve().parent.parent    /    "farfan_pipeline"    /
"phases").resolve())
]

__all__ = [
    "Phase_zero",
    "Phase_one",
    "Phase_two",
    "Phase_three",
    "Phase_four_five_six_seven",
    "Phase_eight",
    "Phase_nine",
]
```

src/core/__init__.py