src/farfan_pipeline/core/observability/__init__.py

"""Core observability utilities."""

```
src/farfan_pipeline/core/observability/structured_logging.py

"""Structured logging helpers.

These are thin utilities used by the verified runner.
"""

from __future__ import annotations

from typing import Any

try:
    import structlog

    _logger: Any = structlog.get_logger(__name__)
except Exception:  # pragma: no cover
    import logging

    _logger = logging.getLogger(__name__)


def log_runtime_config_loaded(*, config_repr: str, runtime_mode: Any) -> None:
    """Emit a single structured event when runtime config loads."""

    # Works with structlog or stdlib logging.
    try:
        _logger.info(
            "runtime_config_loaded",
            runtime_mode=str(getattr(runtime_mode, "value", runtime_mode)),
            config=config_repr,
        )
    except TypeError:
        _logger.info(
            "runtime_config_loaded runtime_mode=%s config=%s",
            str(getattr(runtime_mode, "value", runtime_mode)),
            config_repr,
        )
```

src/farfan_pipeline/core/parameters.py

```
"""
Parameter Loader V2 - Stub for calibration system
"""

from typing import Any


class ParameterLoaderV2:
    """Stub parameter loader for compatibility"""

    @staticmethod
    def get(method_id: str, param_name: str, default: Any) -> Any:
        """Get parameter value, returning default"""
        return default
```

src/farfan_pipeline/core/phases/__init__.py

"""Core phase boundary types."""

src/farfan_pipeline/core/phases/phase2_types.py

```
"""Phase 2 boundary types/validators.

The verified runner expects a validator entrypoint. This is intentionally minimal
until Phase 2 contracts are formalized in one location.
"""

from __future__ import annotations

from typing import Any


def validate_phase2_result(_result: Any) -> bool:
    """Placeholder validator for Phase 2 results.

    Returns True for now (Phase 2 calibration/parametrization removed).
    """

    return True
```

src/farfan_pipeline/core/policy_area_canonicalization.py

```
from __future__ import annotations

import json
import re
from dataclasses import dataclass
from functools import lru_cache
from pathlib import Path
from typing import Final

_LEGACY_POLICY_AREA_RE: Final[re.Pattern[str]] = re.compile(r"^P(?:10|[1-9])$")
_CANONICAL_POLICY_AREA_RE: Final[re.Pattern[str]] = re.compile(r"^PA(?:0[1-9]|10)$")

_REPO_ROOT: Final[Path] = Path(__file__).resolve().parents[3]
_MAPPING_PATH: Final[Path] = _REPO_ROOT / "policy_area_mapping.json"


class PolicyAreaCanonicalizationError(ValueError):
    pass


@dataclass(frozen=True, slots=True)
class PolicyAreaMappingEntry:
    legacy_id: str
    canonical_id: str
    canonical_name: str
    source_of_truth: str


def is_legacy_policy_area_id(value: str) -> bool:
    return _LEGACY_POLICY_AREA_RE.fullmatch(value) is not None


def is_canonical_policy_area_id(value: str) -> bool:
    return _CANONICAL_POLICY_AREA_RE.fullmatch(value) is not None


@lru_cache(maxsize=1)
def _load_policy_area_mapping_entries() -> tuple[PolicyAreaMappingEntry, ...]:
    try:
        payload = json.loads(_MAPPING_PATH.read_text(encoding="utf-8"))
    except FileNotFoundError as exc:
        msg = f"Missing policy area mapping file: {_MAPPING_PATH}"
        raise PolicyAreaCanonicalizationError(msg) from exc
    except json.JSONDecodeError as exc:
        msg = f"Invalid JSON in policy area mapping file: {_MAPPING_PATH}"
        raise PolicyAreaCanonicalizationError(msg) from exc

    if not isinstance(payload, list):
        msg = "policy_area_mapping.json must be a JSON list"
        raise PolicyAreaCanonicalizationError(msg)

    entries: list[PolicyAreaMappingEntry] = []
```

```
    for idx, item in enumerate(payload):
        if not isinstance(item, dict):
            msg = f"Mapping entry {idx} must be an object"
            raise PolicyAreaCanonicalizationError(msg)

        legacy_id = item.get("legacy_id")
        canonical_id = item.get("canonical_id")
        canonical_name = item.get("canonical_name")
        source_of_truth = item.get("source_of_truth")

        if not isinstance(legacy_id, str) or not is_legacy_policy_area_id(legacy_id):
            msg = f"Mapping entry {idx} has invalid legacy_id: {legacy_id!r}"
            raise PolicyAreaCanonicalizationError(msg)

                                if    not    isinstance(canonical_id,    str)    or    not
is_canonical_policy_area_id(canonical_id):
            msg = f"Mapping entry {idx} has invalid canonical_id: {canonical_id!r}"
            raise PolicyAreaCanonicalizationError(msg)

        if not isinstance(canonical_name, str) or not canonical_name.strip():
            msg = f"Mapping entry {idx} has invalid canonical_name: {canonical_name!r}"
            raise PolicyAreaCanonicalizationError(msg)

        if source_of_truth != "monolith":
                            msg  =  f"Mapping  entry  {idx}  has  invalid  source_of_truth:
{source_of_truth!r}"
            raise PolicyAreaCanonicalizationError(msg)

        entries.append(
            PolicyAreaMappingEntry(
                legacy_id=legacy_id,
                canonical_id=canonical_id,
                canonical_name=canonical_name,
                source_of_truth=source_of_truth,
            )
        )

    legacy_ids = [e.legacy_id for e in entries]
    canonical_ids = [e.canonical_id for e in entries]

    if len(set(legacy_ids)) != len(legacy_ids):
        msg = "policy_area_mapping.json contains duplicate legacy_id entries"
        raise PolicyAreaCanonicalizationError(msg)

    if len(set(canonical_ids)) != len(canonical_ids):
        msg = "policy_area_mapping.json contains duplicate canonical_id entries"
        raise PolicyAreaCanonicalizationError(msg)

    return tuple(sorted(entries, key=lambda e: e.canonical_id))


@lru_cache(maxsize=1)
def _legacy_to_canonical_map() -> dict[str, str]:
    return {e.legacy_id: e.canonical_id for e in _load_policy_area_mapping_entries()}
```

```
@lru_cache(maxsize=1)
def _canonical_to_name_map() -> dict[str, str]:
                    return     {e.canonical_id:     e.canonical_name     for     e     in
_load_policy_area_mapping_entries()}


def canonicalize_policy_area_id(value: str) -> str:
    if is_canonical_policy_area_id(value):
        if value in _canonical_to_name_map():
            return value
             raise  PolicyAreaCanonicalizationError(f"Unknown  canonical  policy  area  id:
{value}")

    if is_legacy_policy_area_id(value):
        mapped = _legacy_to_canonical_map().get(value)
        if mapped is None:
                 raise  PolicyAreaCanonicalizationError(f"Unknown  legacy  policy  area  id:
{value}")
        return mapped

    raise PolicyAreaCanonicalizationError(
        f"Invalid policy area id: {value!r} (expected legacy or canonical id)"
    )


def canonical_policy_area_name(value: str) -> str:
    canonical_id = canonicalize_policy_area_id(value)
    return _canonical_to_name_map()[canonical_id]
```

```
src/farfan_pipeline/core/types.py

"""
Core types for F.A.R.F.A.N Policy Analysis Pipeline - VERSIÓN DEFINITIVA PDET

Tipos canónicos para el análisis de Planes de Desarrollo Territorial (PDT) colombianos,
optimizados para municipios PDET (Programas de Desarrollo con Enfoque Territorial).

Basados en:
- Ley 152 de 1994 (Ley Orgánica del Plan de Desarrollo)
- Metodología DNP (Departamento Nacional de Planeación)
- Acuerdo de Paz - Reforma Rural Integral (RRI)
- Plan Marco de Implementación (PMI)
- questionnaire_schema.json v2.0.0
- questionnaire_monolith.json (300 micro-questions)
- pattern_registry.json
- reporte_unit_of_analysis.json

Jerarquía estructural PDT:
    PDT ? Título ? Línea Estratégica ? Sector ? Programa ? Producto ? Meta ? SubMeta

Jerarquía de preguntas (questionnaire):
    MacroQuestion ? MesoQuestion ? MicroQuestion (300)

Dimensiones causales (Cadena de valor DNP):
       D1_Insumos ? D2_Actividades ? D3_Productos ? D4_Resultados ? D5_Impactos ?
D6_Causalidad

Policy Areas (PA01-PA10):
    Derechos humanos específicos del contexto colombiano y construcción de paz
"""

from __future__ import annotations

from dataclasses import dataclass, field
from datetime import datetime
from enum import Enum, auto
from typing import Any, Dict, List, Optional, Tuple, Union
import re


# ============================================================================
# ENUMS - Clasificadores del dominio PDET
# ============================================================================

class NivelJerarquico(Enum):
    """
    Nivel jerárquico en la estructura del PDT.
    Refleja la organización real de los documentos PDM/PDET.
    """
     H1_TITULO = auto()                  # CAPÍTULO / TÍTULO (ej: CAPÍTULO 5. BUENOS
AIRES...)
    H2_LINEA_ESTRATEGICA = auto()   # Línea Estratégica / Eje Estratégico
    H3_SECTOR = auto()              # Sector (ej: 41 Inclusión Social)
```

```
    H4_PROGRAMA = auto()            # Programa Presupuestal
    H5_PRODUCTO = auto()            # Producto / Proyecto MGA
    H6_META = auto()                # Meta Cuatrienio / Indicador
    H7_SUBMETA = auto()             # Metas anuales desagregadas (2024-2027)


class SeccionPDT(Enum):
    """
    Secciones principales del PDT según Ley 152/1994 y estructura real PDET.
    Incluye secciones obligatorias y contextuales.
    """
    # === TÍTULO I: Fundamentos y Componente General ===
    FUNDAMENTOS = "fundamentos"                  # Marco normativo y alineación
    METODOLOGIA = "metodologia"                  # Participación ciudadana, SisPT
    DIAGNOSTICO = "diagnostico"                   # Caracterización territorial, Análisis
de brechas

    # === TÍTULO II: Parte Estratégica ===
    PRINCIPIOS = "principios"                    # Visión, Misión, Principios Rectores
    ENFOQUES = "enfoques"                        # Enfoques Transversales
    PARTE_ESTRATEGICA = "parte_estrategica"      # Líneas/Ejes, Programas, Metas

    # === TÍTULO III: Componente Financiero ===
    PLAN_FINANCIERO = "plan_financiero"          # Diagnóstico fiscal, MFMP
    PLAN_INVERSIONES = "plan_inversiones"        # PPI - Matriz presupuestaria

    # === Capítulos Especiales (Obligatorios/Contextuales) ===
      CAPITULO_PAZ = "capitulo_paz"                       # Construcción de paz / PDET
(obligatorio)
    CAPITULO_SGR = "capitulo_sgr"                # Inversiones SGR (independiente)

    # === TÍTULO IV: Seguimiento y Evaluación ===
    SEGUIMIENTO = "seguimiento"                  # Indicadores, Plan de Acción, SisPT

    # === Otros ===
    ANEXOS = "anexos"                            # Matrices, tablas complementarias


class DimensionCausal(Enum):
    """
    Dimensiones de la cadena de valor DNP.
    Alineado con canonical_notation.dimensions del questionnaire_monolith.json
    Refleja la lógica de intervención pública territorial.
    """
    DIM01_INSUMOS = "DIM01"       # D1: Diagnóstico, Recursos, Capacidad institucional
    DIM02_ACTIVIDADES = "DIM02"   # D2: Diseño de Intervención, Procesos operativos
       DIM03_PRODUCTOS = "DIM03"         # D3: Productos y Outputs (bienes/servicios
entregados)
       DIM04_RESULTADOS = "DIM04"       # D4: Resultados y Outcomes (efectos/cambios
generados)
      DIM05_IMPACTOS = "DIM05"         # D5: Impactos de Largo Plazo (transformación
territorial)
    DIM06_CAUSALIDAD = "DIM06"    # D6: Teoría de Cambio (coherencia cadena de valor)
```

```
    @classmethod
    def from_legacy(cls, legacy_id: str) -> "DimensionCausal":
        """Convierte D1-D6 a DIM01-DIM06."""
        mapping = {
            "D1": cls.DIM01_INSUMOS,
            "D2": cls.DIM02_ACTIVIDADES,
            "D3": cls.DIM03_PRODUCTOS,
            "D4": cls.DIM04_RESULTADOS,
            "D5": cls.DIM05_IMPACTOS,
            "D6": cls.DIM06_CAUSALIDAD,
        }
        return mapping.get(legacy_id, cls.DIM01_INSUMOS)


class CategoriaCausal(Enum):
    """
    Categorías causales jerárquicas para teoría de cambio.
    Axioma: 1-5 en orden INSUMOS->CAUSALIDAD.
    """
    INSUMOS = 1
    PROCESOS = 2
    PRODUCTOS = 3
    RESULTADOS = 4
    CAUSALIDAD = 5


class PolicyArea(Enum):
    """
    Áreas de política del questionnaire (PA01-PA10).
    Alineado con canonical_notation.policy_areas del questionnaire_monolith.json
    Enfoque de derechos humanos en contexto colombiano y PDET.
    """
    PA01 = "PA01"  # Derechos de las mujeres e igualdad de género
    PA02 = "PA02"  # Prevención de la violencia y protección frente al conflicto armado
    PA03 = "PA03"  # Ambiente sano, cambio climático, prevención y atención a desastres
    PA04 = "PA04"  # Derechos económicos, sociales y culturales
    PA05 = "PA05"  # Derechos de las víctimas y construcción de paz
      PA06 = "PA06"   # Derecho al buen futuro de la niñez, adolescencia, juventud y
entornos protectores
    PA07 = "PA07"  # Tierras y territorios
    PA08 = "PA08"  # Líderes y lideresas, defensores y defensoras de derechos humanos
    PA09 = "PA09"  # Crisis de derechos de personas privadas de la libertad
    PA10 = "PA10"  # Migración transfronteriza

    @classmethod
    def from_legacy(cls, legacy_id: str) -> "PolicyArea":
        """Convert legacy policy area id into canonical PolicyArea."""
        from farfan_pipeline.core.policy_area_canonicalization import (
            canonicalize_policy_area_id,
            is_legacy_policy_area_id,
        )

        if not is_legacy_policy_area_id(legacy_id):
            raise ValueError("Invalid legacy policy area id")
```

```
        return cls(canonicalize_policy_area_id(legacy_id))

    @classmethod
    def canonicalize(cls, policy_area_id: str) -> "PolicyArea":
        """Parse legacy or canonical id into canonical PolicyArea."""
                        from   farfan_pipeline.core.policy_area_canonicalization   import
canonicalize_policy_area_id

        return cls(canonicalize_policy_area_id(policy_area_id))


class MarcadorContextual(Enum):
    """
    Marcadores P-D-Q para clasificación de contenido según contexto semántico.
    Permite identificar el tipo de información en el texto del PDT.
    """
    P_PROBLEMA = "problema"      # Diagnóstico, brechas, necesidades, problemáticas
    D_DECISION = "decision"      # Elección estratégica, priorización, objetivos
    Q_PREGUNTA = "pregunta"      # Investigación, evaluación, seguimiento


class TipoIndicador(Enum):
    """
    Clasificación de indicadores según metodología DNP.
    Refleja los tipos reales encontrados en matrices de indicadores PDT.
    """
    PRODUCTO = "producto"        # Bienes/servicios entregados (outputs)
    RESULTADO = "resultado"      # Efectos/cambios generados (outcomes)
    BIENESTAR = "bienestar"      # Calidad de vida, cierre de brechas
    GESTION = "gestion"          # Procesos internos, eficiencia administrativa


class FuenteFinanciacion(Enum):
    """
    Fuentes de financiación en el sistema territorial colombiano.
    Refleja las columnas reales del Plan Plurianual de Inversiones (PPI).
    """
    SGP = "sgp"                          # Sistema General de Participaciones
    SGR = "sgr"                          # Sistema General de Regalías
    RECURSOS_PROPIOS = "recursos_propios"  # Ingresos tributarios locales
    CREDITO = "credito"                  # Endeudamiento público
    COOPERACION = "cooperacion"          # Cooperación internacional
    OTRAS = "otras"                      # Cofinanciación, transferencias
    FONDO_SUBREGIONAL = "fondo_subregional"  # Ej: Alto Patía


class NivelConfianza(Enum):
    """
    Nivel de confianza en la extracción y análisis de datos.
    Basado en umbrales de scoring y calidad de evidencia.
    """
    ALTA = "alta"       # 0.9-1.0 - Datos explícitos, verificados, cuantitativos
    MEDIA = "media"     # 0.6-0.8 - Datos inferidos, parciales, cualitativos
```

```
    BAJA = "baja"       # <0.6 - Datos ausentes, ambiguos o "S/D"


class ScoringLevel(Enum):
    """
    Niveles de scoring del questionnaire.
    Alineado con scoring.micro_levels del questionnaire_schema.json
    """
    EXCELENTE = "excelente"      # min_score: 0.85
    BUENO = "bueno"              # min_score: 0.70
    ACEPTABLE = "aceptable"      # min_score: 0.55
    INSUFICIENTE = "insuficiente"  # min_score: 0.0

    @classmethod
    def from_score(cls, score: float) -> "ScoringLevel":
        """Determina nivel de scoring basado en puntaje."""
        if score >= 0.85:
            return cls.EXCELENTE
        elif score >= 0.70:
            return cls.BUENO
        elif score >= 0.55:
            return cls.ACEPTABLE
        return cls.INSUFICIENTE


class PatternMatchType(Enum):
    """
    Tipos de matching de patrones.
    Alineado con PatternItem.match_type del questionnaire_schema.json
    """
    REGEX = "REGEX"
    LITERAL = "LITERAL"
    NER_OR_REGEX = "NER_OR_REGEX"


class PatternSpecificity(Enum):
    """
    Especificidad de patrones de detección.
    Alineado con PatternItem.specificity del questionnaire_schema.json
    """
    HIGH = "HIGH"      # Patrones muy específicos (ej: códigos MGA)
    MEDIUM = "MEDIUM"  # Patrones moderados (ej: términos técnicos)
    LOW = "LOW"        # Patrones genéricos (ej: palabras comunes)


class PatternContextScope(Enum):
    """
    Alcance de contexto para patrones de detección.
    Alineado con PatternItem.context_scope del questionnaire_schema.json
    """
    SENTENCE = "SENTENCE"      # Oración individual
    PARAGRAPH = "PARAGRAPH"    # Párrafo completo
    SECTION = "SECTION"        # Sección del documento
    DOCUMENT = "DOCUMENT"      # Documento completo
```

```
class MethodType(Enum):
    """
    Tipos de métodos analíticos.
    Alineado con MethodSet.method_type del questionnaire_schema.json
    """
    ANALYSIS = "analysis"          # Análisis de contenido
    AGGREGATION = "aggregation"    # Agregación de resultados
    ROUTING = "routing"            # Enrutamiento de preguntas
    VALIDATION = "validation"      # Validación de datos
    EXTRACTION = "extraction"      # Extracción de información
    SCORING = "scoring"            # Cálculo de puntajes


class AggregationMethod(Enum):
    """
    Métodos de agregación para preguntas multinivel.
    Alineado con MacroQuestion.aggregation_method del questionnaire_schema.json
    """
    HOLISTIC_ASSESSMENT = "holistic_assessment"  # Evaluación holística cualitativa
    WEIGHTED_AVERAGE = "weighted_average"        # Promedio ponderado
    HIERARCHICAL = "hierarchical"                # Agregación jerárquica


class TipoEntidadInstitucional(Enum):
    """
    Tipos de entidades institucionales en el sistema colombiano.
    Refleja la estructura real del Estado y actores territoriales.
    """
    NACIONAL = "nacional"              # DNP, Ministerios, Fiscalía, JEP, UARIV
    DEPARTAMENTAL = "departamental"    # Gobernación, CAR
    MUNICIPAL = "municipal"            # Alcaldía, Secretarías, Consejo Municipal
    COOPERACION = "cooperacion"        # Organismos internacionales
    SOCIEDAD_CIVIL = "sociedad_civil"  # JAC, Consejos Comunitarios, Mesas
    CONTROL = "control"                # Contraloría, Personería


class TipoReferenciaLegal(Enum):
    """
    Tipos de referencias legales en el ordenamiento colombiano.
    """
    CONSTITUCION = "constitucion"  # Constitución Política (Art. 339)
    LEY = "ley"                    # Ley 152 de 1994, Ley 2056 de 2020
    DECRETO = "decreto"            # Decreto 111 de 1996, Decreto 413 de 2018
    RESOLUCION = "resolucion"      # Resolución DNP
    ACUERDO = "acuerdo"            # Acuerdo Municipal
    CIRCULAR = "circular"          # Circular conjunta
    SENTENCIA = "sentencia"        # Sentencias de Cortes


class ZonaPDET(Enum):
    """
    Subregiones PDET (Programas de Desarrollo con Enfoque Territorial).
```

```
    16 subregiones priorizadas del Acuerdo de Paz.
    """
    # Cauca
    ALTO_PATIA_NORTE_CAUCA = "alto_patia_norte_cauca"
    PACIFICO_MEDIO = "pacifico_medio"

    # Otras regiones PDET (ejemplos)
    ARAUCA = "arauca"
    BAJO_CAUCA_NORDESTE_ANTIOQUIA = "bajo_cauca_nordeste_antioquia"
    CATATUMBO = "catatumbo"
    CHOCO = "choco"
    CUENCA_CAGUAN_PUTUMAYO = "cuenca_caguan_putumayo"
    MACARENA_GUAVIARE = "macarena_guaviare"
    MONTES_MARIA = "montes_maria"
    PACIFICO_NARINO = "pacifico_narino"
    PACÍFICO_SUR = "pacifico_sur"
    PUTUMAYO = "putumayo"
    SUR_BOLIVAR = "sur_bolivar"
    SUR_CORDOBA = "sur_cordoba"
    SUR_TOLIMA = "sur_tolima"
    URABÁ_ANTIOQUEÑO = "uraba_antioqueno"


class PilarRRI(Enum):
    """
    Pilares de la Reforma Rural Integral (RRI) - Acuerdo de Paz.
    Los 8 pilares PDET vinculados al capítulo de paz del PDT.
    """
    PILAR_1_ORDENAMIENTO = "pilar_1_ordenamiento_social_propiedad"
    PILAR_2_INFRAESTRUCTURA = "pilar_2_infraestructura_adecuacion_tierras"
    PILAR_3_SALUD = "pilar_3_salud_rural"
    PILAR_4_EDUCACION = "pilar_4_educacion_rural"
    PILAR_5_VIVIENDA = "pilar_5_vivienda_agua_saneamiento"
    PILAR_6_REACTIVACION = "pilar_6_reactivacion_economica_produccion_agropecuaria"
    PILAR_7_RECONCILIACION = "pilar_7_reconciliacion_convivencia_paz"
    PILAR_8_SISTEMA_ALIMENTARIO = "pilar_8_sistema_alimentacion_nutricional"


# ============================================================================
# TIPOS BASE - Provenance y Chunks
# ============================================================================

@dataclass
class Provenance:
    """
    Rastrea el origen y transformación de datos.
    Permite auditoría completa desde el PDF original hasta el análisis final.

    CRÍTICO: Todos los datos extraídos deben tener provenance para trazabilidad.
    """
    source_file: str
    page_number: Optional[int] = None
    chunk_id: Optional[str] = None
    extraction_method: str = "unknown"
```

```
    timestamp: Optional[datetime] = None

    # Localización semántica en el documento
    seccion_pdt: Optional[SeccionPDT] = None
    nivel_jerarquico: Optional[NivelJerarquico] = None

    # Offsets para trazabilidad exacta (posición en el texto)
    start_offset: Optional[int] = None
    end_offset: Optional[int] = None

    # Confianza en la extracción
    confidence_score: float = 0.0
    nivel_confianza: NivelConfianza = NivelConfianza.BAJA

    # Contexto adicional
    metadata: Dict[str, Any] = field(default_factory=dict)

    def to_dict(self) -> Dict[str, Any]:
        """Serializa provenance a diccionario."""
        return {
            "source_file": self.source_file,
            "page_number": self.page_number,
            "chunk_id": self.chunk_id,
            "extraction_method": self.extraction_method,
            "timestamp": self.timestamp.isoformat() if self.timestamp else None,
            "seccion_pdt": self.seccion_pdt.value if self.seccion_pdt else None,
                "nivel_jerarquico": self.nivel_jerarquico.value if self.nivel_jerarquico
else None,
            "start_offset": self.start_offset,
            "end_offset": self.end_offset,
            "confidence_score": self.confidence_score,
            "nivel_confianza": self.nivel_confianza.value,
            "metadata": self.metadata
        }


@dataclass
class TextSpan:
    """
    Representa un span de texto con posición exacta.
    Útil para resaltar evidencias en el documento original.
    """
    text: str
    start: int
    end: int
    page: Optional[int] = None


@dataclass
class ChunkData:
    """
    Unidad mínima de texto extraído con contexto completo.

    Cada chunk preserva:
```

```
    - Texto original sin modificar
    - Posición exacta (offsets, página)
    - Clasificación semántica (sección, nivel, dimensión causal, policy area)
    - Provenance completo
    - Relaciones jerárquicas (parent/children)

    INVARIANTE CRÍTICO: El sistema debe generar exactamente 60 chunks base:
    10 Policy Areas × 6 Dimensions = 60 chunks
    """
    chunk_id: str
    text: str
    start_offset: int
    end_offset: int

    # Localización física
    page_number: Optional[int] = None
    seccion_pdt: Optional[SeccionPDT] = None
    nivel_jerarquico: Optional[NivelJerarquico] = None

    # Clasificación semántica (coordenadas en el espacio de análisis)
    dimension_causal: Optional[DimensionCausal] = None
    policy_area: Optional[PolicyArea] = None
    marcador_contextual: Optional[MarcadorContextual] = None

    # Trazabilidad
    provenance: Optional[Provenance] = None

    # Relaciones jerárquicas
    parent_chunk_id: Optional[str] = None
    child_chunk_ids: List[str] = field(default_factory=list)

    # Metadatos adicionales
    metadata: Dict[str, Any] = field(default_factory=dict)

    def get_coordinate(self) -> Tuple[str, str]:
        """
        Retorna la coordenada (PolicyArea, Dimension) del chunk.
        Útil para indexación y búsqueda.
        """
        pa = self.policy_area.value if self.policy_area else "UNKNOWN"
        dim = self.dimension_causal.value if self.dimension_causal else "UNKNOWN"
        return (pa, dim)


# ============================================================================
# TIPOS DE QUESTIONNAIRE - Alineados con questionnaire_schema.json
# ============================================================================

@dataclass
class PatternItem:
    """
    Patrón de detección para análisis de texto.
    Alineado con definitions.PatternItem del questionnaire_schema.json
```

```
    Los patrones son la base del análisis automatizado de PDTs.
    """
    id: str
    pattern: str
    match_type: PatternMatchType
    category: str

    # Contexto semántico
    context_requirement: Optional[str] = None
    context_scope: PatternContextScope = PatternContextScope.PARAGRAPH

    # Especificidad y expansión
    specificity: PatternSpecificity = PatternSpecificity.MEDIUM
    semantic_expansion: Optional[Union[str, Dict, List]] = None
    synonym_clusters: Optional[List[str]] = None

    # Validación y confianza
    validation_rule: Optional[str] = None
    confidence_weight: Optional[float] = None
    negative_filter: Optional[bool] = None

    # Tipo de entidad
    entity_type: Optional[str] = None
    element_tags: Optional[List[str]] = None

    # Referencia a patrón compartido (pattern_registry.json)
    pattern_ref: Optional[str] = None  # e.g., "PAT-0001"

    def matches(self, text: str, context: Optional[str] = None) -> bool:
        """
        Verifica si el patrón hace match con el texto.
        Considera el contexto si es requerido.
        """
        if self.match_type == PatternMatchType.LITERAL:
            return self.pattern.lower() in text.lower()
        elif self.match_type == PatternMatchType.REGEX:
            return bool(re.search(self.pattern, text, re.IGNORECASE))
        elif self.match_type == PatternMatchType.NER_OR_REGEX:
            # Requiere integración con NER, por ahora fallback a regex
            return bool(re.search(self.pattern, text, re.IGNORECASE))
        return False


@dataclass
class MethodSet:
    """
    Conjunto de métodos analíticos para una pregunta.
    Alineado con definitions.MethodSet del questionnaire_schema.json

    Define la estrategia de análisis para responder una micro-question.
    """
    class_name: str  # Pattern: ^[A-Za-z_][A-Za-z0-9_]*$
    function: str    # Pattern: ^[A-Za-z_][A-Za-z0-9_]*$
    method_type: MethodType
```

```
    priority: int = 0
    description: Optional[str] = None

    # Dependencias de patrones
    depends_on_patterns: List[str] = field(default_factory=list)
    pattern_dependencies: List[str] = field(default_factory=list)

    # Output esperado
    produces_elements: List[str] = field(default_factory=list)
    output_validates: List[str] = field(default_factory=list)

    failure_mode: Optional[str] = None


@dataclass
class MicroQuestion:
    """
    Pregunta micro del cuestionario (300 preguntas).
    Alineado con definitions.MicroQuestion del questionnaire_schema.json

    Representa la unidad atómica de análisis del PDT.
    Cada micro-question evalúa un aspecto específico de la calidad del plan.
    """
    question_id: str  # Pattern: ^Q\d{3}$ (Q001-Q300)
    text: str
    cluster_id: str
    dimension_id: str  # DIM01-DIM06

    policy_area_id: Optional[str] = None  # PA01-PA10
    base_slot: Optional[str] = None
    question_global: Optional[int] = None
    scoring_modality: Optional[str] = None
    scoring_definition_ref: Optional[str] = None

    # Elementos esperados en la respuesta
    expected_elements: List[str] = field(default_factory=list)

    # Contrato de fallo (qué hacer si no se puede responder)
    failure_contract: Optional[Dict[str, Any]] = None

    # Métodos y patrones asociados
    method_sets: List[MethodSet] = field(default_factory=list)
    patterns: List[PatternItem] = field(default_factory=list)

    # Validaciones
    validations: Optional[Dict[str, Any]] = None

    def validate_question_id(self) -> bool:
        """Valida que question_id siga el patrón ^Q\\d{3}$."""
        return bool(re.match(r"^Q\d{3}$", self.question_id))


@dataclass
```

```
class MesoQuestion:
    """
    Pregunta meso del cuestionario (clusters temáticos).
    Alineado con definitions.MesoQuestion del questionnaire_schema.json

    Agrega múltiples micro-questions relacionadas temáticamente.
    """
    question_id: str  # Pattern: ^MESO_[A-Z0-9_]+$
    cluster_id: str
    text: str
    type: str

    question_global: Optional[int] = None
    aggregation_method: Optional[str] = None
    scoring_modality: Optional[str] = None
    policy_areas: List[str] = field(default_factory=list)
    patterns: List[Dict[str, Any]] = field(default_factory=list)

    # Micro questions que agrega
    micro_question_ids: List[str] = field(default_factory=list)


@dataclass
class MacroQuestion:
    """
    Pregunta macro del cuestionario (evaluación holística).
    Alineado con definitions.MacroQuestion del questionnaire_schema.json

    Evaluación de más alto nivel que integra múltiples meso-questions.
    """
    question_global: int
    aggregation_method: AggregationMethod

    clusters: List[str] = field(default_factory=list)
    patterns: List[Dict[str, Any]] = field(default_factory=list)
    fallback: Optional[Dict[str, Any]] = None


@dataclass
class ScoringDefinition:
    """
    Definición de scoring para un nivel de evaluación.
    Vincula puntajes numéricos con categorías cualitativas.
    """
    level: ScoringLevel
    min_score: float
    criteria: str = ""
    color: Optional[str] = None  # Para visualización


# ============================================================================
# TIPOS DE DOMINIO PDT - Estructura Jerárquica del Territorio
# ============================================================================
```

```python
@dataclass
class EntidadInstitucional:
    """
    Entidad institucional colombiana referenciada en el PDT.
    Refleja el ecosistema de actores del sistema territorial.
    """
    nombre: str
    sigla: Optional[str] = None
    tipo: TipoEntidadInstitucional = TipoEntidadInstitucional.MUNICIPAL
    rol: Optional[str] = None

    # Información de contacto/ubicación (opcional)
    nivel_gobierno: Optional[str] = None  # nacional, departamental, municipal
    metadata: Dict[str, Any] = field(default_factory=dict)


@dataclass
class ReferenciaLegal:
    """
    Referencia a normativa colombiana citada en el PDT.
    Permite rastrear el marco legal de las intervenciones.
    """
    tipo: TipoReferenciaLegal
    numero: str
    año: int
    articulo: Optional[str] = None
    descripcion: Optional[str] = None

    # Texto completo de la referencia (ej: "Ley 152 de 1994")
    texto_completo: Optional[str] = None


@dataclass
class Meta:
    """
    Meta cuantificable del PDT.
    Vincula indicador + línea base + objetivo cuatrienal.

    Refleja la estructura real de las Matrices de Indicadores.
    """
    codigo: Optional[str] = None
    descripcion: str = ""

    # Indicador asociado
    tipo_indicador: TipoIndicador = TipoIndicador.PRODUCTO
    indicador_nombre: str = ""
    codigo_indicador: Optional[str] = None
    unidad_medida: str = ""

    # Valores (Línea Base ? Meta Cuatrienio)
    linea_base: Optional[float] = None
    año_linea_base: Optional[int] = None
    meta_cuatrienio: Optional[float] = None
```

```
    # Desagregación anual (submetas)
    meta_2024: Optional[float] = None
    meta_2025: Optional[float] = None
    meta_2026: Optional[float] = None
    meta_2027: Optional[float] = None

     # Información adicional
    fuente_informacion: Optional[str] = None
    responsable: Optional[str] = None  # Secretaría responsable

    # Grupo poblacional objetivo
    poblacion_objetivo: Optional[str] = None

    # Trazabilidad
    provenance: Optional[Provenance] = None

    def cumplimiento_porcentual(self) -> Optional[float]:
        """Calcula % de cumplimiento si hay línea base y meta."""
        if self.linea_base is not None and self.meta_cuatrienio is not None:
            if self.meta_cuatrienio == 0:
                return None
            return (self.linea_base / self.meta_cuatrienio) * 100
        return None


@dataclass
class Producto:
    """
    Producto del PDT (bien o servicio entregable).
    Codificado según catálogo MGA (Metodología General Ajustada) del DNP.

    Representa el nivel H5 de la jerarquía PDT.
    """
    codigo_mga: Optional[str] = None  # Código del catálogo MGA (ej: 2106003)
    nombre: str = ""
    descripcion: str = ""

    # Indicadores y metas asociadas
    indicadores: List[Meta] = field(default_factory=list)

    # Presupuesto del producto
    costo_total: Optional[float] = None
    fuentes_financiacion: Dict[FuenteFinanciacion, float] = field(default_factory=dict)

    # Información adicional
        alcance_geografico:  Optional[str]  =  None    #  Rural,  Urbano,  Corregimientos
específicos
    poblacion_beneficiaria: Optional[str] = None

    # Trazabilidad
    provenance: Optional[Provenance] = None


@dataclass
```

```
class Programa:
    """
    Programa presupuestal del PDT.
    Agrupa productos relacionados bajo un código presupuestal único.

    Representa el nivel H4 de la jerarquía PDT.
    Refleja la estructura real de los documentos (ej: "Programa: Salud Pública").
    """
    codigo_presupuestal: Optional[str] = None  # Código del programa (ej: 1203)
    nombre: str = ""
    descripcion: str = ""
    sector: Optional[str] = None  # Ej: "41 Inclusión Social"

    # Contenido del programa
    productos: List[Producto] = field(default_factory=list)
    metas_resultado: List[Meta] = field(default_factory=list)

    # Presupuesto agregado
    costo_total_cuatrienio: Optional[float] = None
     distribucion_anual: Dict[int, float] = field(default_factory=dict)  # {2024: monto,
...}

    # Justificación (vínculo con diagnóstico)
    justificacion: Optional[str] = None
    problematicas_atendidas: List[str] = field(default_factory=list)

    # Trazabilidad
    provenance: Optional[Provenance] = None


@dataclass
class LineaEstrategica:
    """
    Línea/Eje estratégico del PDT.
    Nivel más alto de organización programática (H2).

    Refleja las "grandes apuestas" de la administración territorial.
    Ej: "Línea Estratégica 1: Territorio, Ambiente y Ruralidad"
    """
    codigo: Optional[str] = None
    numero: Optional[int] = None
    titulo: str = ""
    descripcion: Optional[str] = None

    # Enfoques transversales aplicados
    enfoques: List[str] = field(default_factory=list)
    # Ej: ["derechos", "género", "territorial", "diferencial", "sostenibilidad"]

    # Contenido de la línea
    programas: List[Programa] = field(default_factory=list)

    # Presupuesto agregado de la línea
    inversion_total: Optional[float] = None
    distribucion_anual: Dict[int, float] = field(default_factory=dict)
```

```
    # Vinculación con ODS y PND
    ods_vinculados: List[int] = field(default_factory=list)  # Números de ODS (1-17)
    vinculacion_pnd: Optional[str] = None  # Transformación Colombia Potencia Mundial de la Vida

    # Trazabilidad
    provenance: Optional[Provenance] = None


@dataclass
class DiagnosticoTerritorial:
    """
    Información del diagnóstico/caracterización del PDT (Título I).
    Fundamenta la intervención pública identificando brechas y necesidades.
    """
    # === Datos demográficos ===
    poblacion_total: Optional[int] = None
    poblacion_urbana: Optional[int] = None
    poblacion_rural: Optional[int] = None
    poblacion_etnica: Optional[Dict[str, int]] = None  # {"indigena": X, "afro": Y}

    # === Indicadores socioeconómicos ===
    indice_pobreza_multidimensional: Optional[float] = None  # IPM
    nbi: Optional[float] = None  # Necesidades Básicas Insatisfechas
    cobertura_salud: Optional[Dict[str, int]] = None  # {"contributivo": X, "subsidiado": Y}
    tasas_educacion: Optional[Dict[str, float]] = None  # {"cobertura": X, "desercion": Y}

    # === Problemáticas identificadas ===
    problematicas: List[str] = field(default_factory=list)
    brechas: List[str] = field(default_factory=list)
    ejes_problematicos: List[str] = field(default_factory=list)

    # === División territorial ===
    corregimientos: List[str] = field(default_factory=list)
    veredas: List[str] = field(default_factory=list)
    consejos_comunitarios: List[str] = field(default_factory=list)

    # === Instrumentos de ordenamiento ===
    tiene_pot: bool = False
    tiene_pbot: bool = False
    tiene_eot: bool = False
    fecha_actualizacion_ot: Optional[datetime] = None

    # Trazabilidad
    provenance: Optional[Provenance] = None


@dataclass
class PlanPlurianuaInversiones:
    """
    Plan Plurianual de Inversiones (PPI) - Título III del PDT.
```

```
    Componente financiero obligatorio que proyecta recursos del cuatrienio.

    Refleja la estructura real de las Matrices PPI encontradas en los documentos.
    """
    # === Totales por vigencia (año fiscal) ===
    total_2024: Optional[float] = None
    total_2025: Optional[float] = None
    total_2026: Optional[float] = None
    total_2027: Optional[float] = None
    total_cuatrienio: Optional[float] = None

    # === Desglose por fuente de financiación ===
    total_sgp: Optional[float] = None  # Sistema General de Participaciones
    total_sgr: Optional[float] = None  # Sistema General de Regalías
    total_recursos_propios: Optional[float] = None
    total_credito: Optional[float] = None
    total_cooperacion: Optional[float] = None
    total_otras_fuentes: Optional[float] = None

    # === Matriz de inversiones detallada ===
    # Lista de diccionarios con estructura:
    # {"linea": str, "sector": str, "programa": str, "2024": float, "2025": float, ...}
    matriz_inversiones: List[Dict[str, Any]] = field(default_factory=list)

    # === Análisis fiscal ===
     marco_fiscal_mediano_plazo: Optional[Dict[str, Any]] = None  # MFMP (proyección 10
años)
    sostenibilidad_fiscal: Optional[str] = None  # Análisis cualitativo

    # Trazabilidad
    provenance: Optional[Provenance] = None

    def calcular_porcentaje_fuente(self, fuente: FuenteFinanciacion) -> Optional[float]:
        """Calcula % de una fuente respecto al total."""
        if self.total_cuatrienio is None or self.total_cuatrienio == 0:
            return None

        mapping = {
            FuenteFinanciacion.SGP: self.total_sgp,
            FuenteFinanciacion.SGR: self.total_sgr,
            FuenteFinanciacion.RECURSOS_PROPIOS: self.total_recursos_propios,
            FuenteFinanciacion.CREDITO: self.total_credito,
            FuenteFinanciacion.COOPERACION: self.total_cooperacion,
            FuenteFinanciacion.OTRAS: self.total_otras_fuentes,
        }

        valor = mapping.get(fuente)
        if valor is not None:
            return (valor / self.total_cuatrienio) * 100
        return None


@dataclass
class IniciativaPDET:
```

```
    """
    Iniciativa específica del PDET vinculada al capítulo de paz.
    Las iniciativas PDET son priorizadas por las comunidades en los PATR
    (Planes de Acción para la Transformación Regional).
    """
    # Clasificación PDET (pilar es obligatorio, los demás campos son opcionales)
    pilar_rri: PilarRRI  # Uno de los 8 pilares
    codigo: Optional[str] = None
    nombre: str = ""
    descripcion: str = ""

    linea_accion: Optional[str] = None

    # Vinculación con el PDT
    programa_vinculado: Optional[str] = None  # Código del programa PDT
    presupuesto_asignado: Optional[float] = None

    # Ubicación
    corregimientos_veredas: List[str] = field(default_factory=list)

    # Trazabilidad
    provenance: Optional[Provenance] = None


@dataclass
class CapituloPaz:
    """
    Capítulo de Paz/PDET - Obligatorio en municipios PDET.

    Articulación del PDT con el Acuerdo de Paz (2016) y el Plan Marco
    de Implementación (PMI). Fundamental en zonas afectadas por el conflicto.

    Refleja la estructura real del "CAPÍTULO 5. BUENOS AIRES ACTÚA POR LA PAZ"
    y secciones similares en documentos PDET.
    """
    # === Clasificación PDET ===
    es_municipio_pdet: bool = False
    subregion_pdet: Optional[ZonaPDET] = None

    # === Iniciativas PDET priorizadas ===
    iniciativas_pdet: List[IniciativaPDET] = field(default_factory=list)
    total_iniciativas_priorizadas: int = 0

    # === Articulación con pilares RRI ===
    pilares_abordados: List[PilarRRI] = field(default_factory=list)

    # === Articulación institucional ===
    articulacion_pmi: Optional[str] = None  # Plan Marco de Implementación
    articulacion_patr: Optional[str] = None  # Plan de Acción Transformación Regional
    articulacion_pnis: Optional[str] = None  # Programa Nacional Integral Sustitución

    # === Presupuesto específico paz ===
    presupuesto_total_paz: Optional[float] = None
    fuentes_paz: Dict[FuenteFinanciacion, float] = field(default_factory=dict)
```

```
    # === Población beneficiaria ===
    victimas_conflicto: Optional[int] = None
    excombatientes: Optional[int] = None
    comunidades_etnicas_beneficiadas: List[str] = field(default_factory=list)

    # === Metas específicas de construcción de paz ===
    metas_reconciliacion: List[Meta] = field(default_factory=list)
    metas_reintegracion: List[Meta] = field(default_factory=list)
    metas_reparacion_victimas: List[Meta] = field(default_factory=list)

    # === Coordinación interinstitucional ===
    entidades_coordinacion: List[EntidadInstitucional] = field(default_factory=list)
    # Ej: ART, Agencia Renovación Territorio; ARN, Agencia Reintegración

    # Trazabilidad
    provenance: Optional[Provenance] = None


# ============================================================================
# DOCUMENTO PREPROCESADO - Output de Phase 1
# ============================================================================

@dataclass
class PreprocessedDocument:
    """
    Documento PDT después de preprocesamiento.

    Output canónico de Phase 1 (Ingestion).
    Input para el Orchestrator y fases subsiguientes.

    INVARIANTE CRÍTICO: chunk_count == 60 (10 Policy Areas × 6 Dimensions)
    Cada chunk debe tener coordenadas (PA, DIM) únicas.
    """
    # === Identificación ===
    document_id: str
    source_path: str
    municipio: Optional[str] = None
    departamento: str = "Cauca"  # Default para el contexto actual
    periodo: str = "2024-2027"

    # === Clasificación PDET ===
    es_municipio_pdet: bool = False
    subregion_pdet: Optional[ZonaPDET] = None

    # === Chunks extraídos (60 unidades mínimas) ===
    chunks: List[ChunkData] = field(default_factory=list)

    # === Estructura jerárquica parseada ===
    diagnostico: Optional[DiagnosticoTerritorial] = None
    lineas_estrategicas: List[LineaEstrategica] = field(default_factory=list)
    plan_inversiones: Optional[PlanPlurianuaInversiones] = None
    capitulo_paz: Optional[CapituloPaz] = None
```

```
# === Entidades y referencias extraídas ===
entidades_mencionadas: List[EntidadInstitucional] = field(default_factory=list)
referencias_legales: List[ReferenciaLegal] = field(default_factory=list)

# === Metadatos de extracción ===
total_pages: int = 0
extraction_timestamp: Optional[datetime] = None
extraction_method: str = "unknown"

# === Métricas de calidad ===
provenance_completeness: float = 0.0
chunks_with_provenance: int = 0
chunks_total: int = 0

# === Validación de invariantes ===
invariant_60_chunks: bool = False
coverage_matrix: Dict[Tuple[str, str], bool] = field(default_factory=dict)
# Matriz PA×DIM: {("PA01", "DIM01"): True, ...}

metadata: Dict[str, Any] = field(default_factory=dict)

def compute_provenance_completeness(self) -> float:
    """Calcula métrica de completitud de provenance."""
    if not self.chunks:
        return 0.0

    with_provenance = sum(1 for c in self.chunks if c.provenance is not None)
    self.chunks_with_provenance = with_provenance
    self.chunks_total = len(self.chunks)
    self.provenance_completeness = with_provenance / len(self.chunks)

    return self.provenance_completeness

def validate_chunk_invariant(self) -> bool:
    """
    Valida el invariante de 60 chunks (10 PA × 6 DIM).
    Retorna True si la estructura es correcta.
    """
    self.invariant_60_chunks = len(self.chunks) == 60

    # Construir matriz de cobertura
    self.coverage_matrix = {}
    for pa_num in range(1, 11):  # PA01-PA10
        for dim_num in range(1, 7):  # DIM01-DIM06
            pa_id = f"PA{pa_num:02d}"
            dim_id = f"DIM{dim_num:02d}"
            coord = (pa_id, dim_id)

            # Verificar si existe chunk con esta coordenada
            has_chunk = any(
                c.policy_area and c.dimension_causal and
                c.policy_area.value == pa_id and c.dimension_causal.value == dim_id
                for c in self.chunks
            )
```

```
                self.coverage_matrix[coord] = has_chunk

        # Verificar cobertura completa
        full_coverage = all(self.coverage_matrix.values())

        return self.invariant_60_chunks and full_coverage

          def  get_chunk_by_coordinate(self,  policy_area:  str,  dimension:  str)  ->
Optional[ChunkData]:
        """Obtiene chunk específico por coordenada (PA, DIM)."""
        for chunk in self.chunks:
            if (chunk.policy_area and chunk.policy_area.value == policy_area and
                chunk.dimension_causal and chunk.dimension_causal.value == dimension):
                return chunk
        return None

    def get_chunks_by_policy_area(self, policy_area: str) -> List[ChunkData]:
        """Obtiene todos los chunks de una Policy Area específica."""
        return [c for c in self.chunks
                if c.policy_area and c.policy_area.value == policy_area]

    def get_chunks_by_dimension(self, dimension: str) -> List[ChunkData]:
        """Obtiene todos los chunks de una Dimensión específica."""
        return [c for c in self.chunks
                if c.dimension_causal and c.dimension_causal.value == dimension]


# ============================================================================
# TIPOS PARA ANÁLISIS - Phases 2+
# ============================================================================

@dataclass
class CausalLink:
    """
    Vínculo causal identificado en el PDT.
    Conecta elementos de la cadena de valor DNP (D1?D2?D3?D4?D5).

    Crítico para evaluar la coherencia lógica del plan (Dimensión D6).
    """
    source_chunk_id: str
    target_chunk_id: str

    dimension_source: DimensionCausal
    dimension_target: DimensionCausal

    # Tipo de relación
    tipo_relacion: str  # "causa", "temporal", "jerarquica", "geografica"
    conector_textual: Optional[str] = None  # Frase que establece el vínculo

    # Ejemplo: "a través de la estructuración y ejecución de proyectos...
    # que contribuyen al logro de las transformaciones"

    # Fortaleza del vínculo
    confidence: float = 0.0
```

```
    explicito: bool = False  # ¿Vínculo explícito o inferido?

    # Trazabilidad
    provenance: Optional[Provenance] = None


@dataclass
class ExtractedEvidence:
    """
    Evidencia extraída para responder una pregunta de análisis.
    Vincula texto específico del PDT con una micro-question del questionnaire.
    """
    question_id: str  # Q001-Q300
    chunk_ids: List[str] = field(default_factory=list)

    # Evidencia textual
    evidencia_textual: str = ""
    evidencia_contraria: Optional[str] = None  # Contraejemplos o inconsistencias

    # Clasificación
    dimension_causal: Optional[DimensionCausal] = None
    policy_area: Optional[PolicyArea] = None
    marcador_contextual: Optional[MarcadorContextual] = None

    # Scoring
    score: float = 0.0
    scoring_level: ScoringLevel = ScoringLevel.INSUFICIENTE
    nivel_confianza: NivelConfianza = NivelConfianza.BAJA

    # Trazabilidad
    provenance: Optional[Provenance] = None


@dataclass
class MicroQuestionResult:
    """
    Resultado de evaluación de una MicroQuestion (Q001-Q300).
    Unidad atómica de análisis del pipeline.
    """
    question_id: str  # Q001-Q300
    question_text: str

    # === Scoring ===
    score: float
    scoring_level: ScoringLevel
    scoring_modality: Optional[str] = None

    # === Evidencias ===
    evidencias: List[ExtractedEvidence] = field(default_factory=list)
    n_evidencias_positivas: int = 0
    n_evidencias_negativas: int = 0

    # === Patrones y métodos aplicados ===
    patterns_matched: List[str] = field(default_factory=list)
```

```
    methods_applied: List[str] = field(default_factory=list)

    # === Contexto ===
    dimension_id: Optional[str] = None  # DIM01-DIM06
    policy_area_id: Optional[str] = None  # PA01-PA10
    cluster_id: Optional[str] = None

    # === Hallazgos específicos ===
    hallazgos: List[str] = field(default_factory=list)
    recomendaciones: List[str] = field(default_factory=list)

    # === Trazabilidad y performance ===
    execution_time_ms: Optional[float] = None
    provenance: Optional[Provenance] = None

    metadata: Dict[str, Any] = field(default_factory=dict)


@dataclass
class MesoQuestionResult:
    """
    Resultado agregado de evaluación de una MesoQuestion (cluster temático).
    Agrega múltiples micro-questions relacionadas.
    """
    question_id: str  # MESO_[ID]
    question_text: str

    # === Scoring agregado ===
    score: float
    scoring_level: ScoringLevel
    aggregation_method: AggregationMethod

    # === Micro questions agregadas ===
    micro_results: List[MicroQuestionResult] = field(default_factory=list)
    n_micro_evaluated: int = 0

    # Distribución de scoring
    distribution: Dict[ScoringLevel, int] = field(default_factory=dict)

    # === Contexto ===
    cluster_id: str = ""
    policy_areas: List[str] = field(default_factory=list)

    # === Hallazgos consolidados ===
    hallazgos: List[str] = field(default_factory=list)
    recomendaciones: List[str] = field(default_factory=list)

    metadata: Dict[str, Any] = field(default_factory=dict)


@dataclass
class MacroQuestionResult:
    """
    Resultado de evaluación holística (MacroQuestion).
```

```
    Nivel más alto de agregación, integra múltiples meso-questions.
    """
    score: float
    scoring_level: ScoringLevel
    aggregation_method: AggregationMethod

    # === Meso questions agregadas ===
    meso_results: List[MesoQuestionResult] = field(default_factory=list)
    n_meso_evaluated: int = 0

    # === Hallazgos globales ===
    hallazgos: List[str] = field(default_factory=list)
    recomendaciones: List[str] = field(default_factory=list)

    # Fortalezas y debilidades identificadas
    fortalezas: List[str] = field(default_factory=list)
    debilidades: List[str] = field(default_factory=list)

    metadata: Dict[str, Any] = field(default_factory=dict)


@dataclass
class AnalysisResult:
    """
    Resultado completo del análisis del pipeline F.A.R.F.A.N.
    Representa el output final de todo el proceso de evaluación.

    Este es el artefacto que se entrega como resultado del análisis.
    """
    # === Identificación ===
    document_id: str
    municipio: str
    departamento: str = "Cauca"
    periodo: str = "2024-2027"
    analysis_timestamp: datetime = field(default_factory=datetime.now)

    # === Clasificación PDET ===
    es_municipio_pdet: bool = False
    subregion_pdet: Optional[ZonaPDET] = None

    # === Resultados por nivel jerárquico ===
    macro_result: Optional[MacroQuestionResult] = None
    meso_results: List[MesoQuestionResult] = field(default_factory=list)
    micro_results: List[MicroQuestionResult] = field(default_factory=list)

    # === Métricas agregadas globales ===
    overall_score: float = 0.0
    overall_level: ScoringLevel = ScoringLevel.INSUFICIENTE

    # === Scoring por dimensión (cadena de valor) ===
    scores_by_dimension: Dict[str, float] = field(default_factory=dict)
    # {"DIM01": 0.75, "DIM02": 0.82, ...}

    # === Scoring por policy area (derechos) ===
```

```
    scores_by_policy_area: Dict[str, float] = field(default_factory=dict)
    # {"PA01": 0.68, "PA05": 0.91, ...}

    # === Matriz de cobertura PA×DIM ===
    coverage_matrix_scores: Dict[Tuple[str, str], float] = field(default_factory=dict)
    # {("PA01", "DIM01"): 0.85, ...}

    # === Hallazgos y recomendaciones consolidadas ===
    hallazgos_principales: List[str] = field(default_factory=list)
    recomendaciones_prioritarias: List[str] = field(default_factory=list)

    fortalezas_principales: List[str] = field(default_factory=list)
    debilidades_principales: List[str] = field(default_factory=list)

    # === Evaluación específica PDET (si aplica) ===
    evaluacion_pdet: Optional[Dict[str, Any]] = None
    # {"articulacion_patr": score, "pilares_rri_cubiertos": [...], ...}

    # === Integridad del análisis ===
    questions_executed: int = 0
    questions_total: int = 300
    execution_success: bool = False
    execution_errors: List[str] = field(default_factory=list)

    # === Hashes para verificación de integridad ===
    input_hash: Optional[str] = None  # Hash del documento de entrada
    output_hash: Optional[str] = None  # Hash de este resultado

    # === Performance ===
    total_execution_time_seconds: Optional[float] = None

    metadata: Dict[str, Any] = field(default_factory=dict)

    def compute_overall_score(self) -> float:
        """
        Calcula el score global del análisis.
        Puede usar diferentes estrategias de agregación.
        """
        if self.macro_result:
            self.overall_score = self.macro_result.score
        elif self.meso_results:
            # Promedio ponderado de meso-questions
                    self.overall_score = sum(m.score for m in self.meso_results) /
len(self.meso_results)
        elif self.micro_results:
            # Promedio simple de micro-questions
                   self.overall_score = sum(m.score for m in self.micro_results) /
len(self.micro_results)

        self.overall_level = ScoringLevel.from_score(self.overall_score)
        return self.overall_score

    def generate_summary_report(self) -> Dict[str, Any]:
        """
```

```
        Genera un resumen ejecutivo del análisis.
        Útil para visualización y reportes.
        """
        return {
            "municipio": self.municipio,
            "departamento": self.departamento,
            "periodo": self.periodo,
            "es_pdet": self.es_municipio_pdet,
            "score_global": self.overall_score,
            "nivel_global": self.overall_level.value,
            "fecha_analisis": self.analysis_timestamp.isoformat(),
            "preguntas_evaluadas": f"{self.questions_executed}/{self.questions_total}",
            "fortalezas": self.fortalezas_principales[:5],  # Top 5
            "debilidades": self.debilidades_principales[:5],  # Top 5
            "recomendaciones": self.recomendaciones_prioritarias[:10],  # Top 10
            "scores_por_dimension": self.scores_by_dimension,
            "scores_por_policy_area": self.scores_by_policy_area,
        }


# ============================================================================
# UTILIDADES Y VALIDACIONES
# ============================================================================

def validate_pdt_structure(doc: PreprocessedDocument) -> Dict[str, Any]:
    """
    Valida la estructura de un PDT preprocesado contra los requisitos.

    Returns:
        Dict con resultados de validación y errores encontrados.
    """
    errors = []
    warnings = []

    # Validar invariante de 60 chunks
    if not doc.validate_chunk_invariant():
        errors.append(f"Invariante de 60 chunks no cumplido. Found: {len(doc.chunks)}")

    # Validar completitud de provenance
    if doc.provenance_completeness < 0.9:
                                    warnings.append(f"Provenance    completeness    bajo:
{doc.provenance_completeness:.2%}")

    # Validar secciones obligatorias
    if not doc.diagnostico:
        errors.append("Falta sección de Diagnóstico (obligatoria)")

    if not doc.lineas_estrategicas:
        errors.append("Faltan Líneas Estratégicas (obligatorias)")

    if not doc.plan_inversiones:
        errors.append("Falta Plan Plurianual de Inversiones (obligatorio)")

    # Validar capítulo PDET si es municipio PDET
```

```
    if doc.es_municipio_pdet and not doc.capitulo_paz:
        errors.append("Municipio PDET sin Capítulo de Paz (obligatorio)")

    return {
        "valid": len(errors) == 0,
        "errors": errors,
        "warnings": warnings,
        "checks_passed": {
            "chunk_invariant": doc.invariant_60_chunks,
            "provenance_complete": doc.provenance_completeness >= 0.9,
            "has_diagnostico": doc.diagnostico is not None,
            "has_estrategia": len(doc.lineas_estrategicas) > 0,
            "has_ppi": doc.plan_inversiones is not None,
                "has_paz_if_pdet": not doc.es_municipio_pdet or doc.capitulo_paz is not
None,
        }
    }


def create_empty_preprocessed_document(
    document_id: str,
    source_path: str,
    municipio: str,
    es_pdet: bool = False
) -> PreprocessedDocument:
    """
    Crea un PreprocessedDocument vacío con la estructura básica.
    Útil para inicializar el pipeline.
    """
    return PreprocessedDocument(
        document_id=document_id,
        source_path=source_path,
        municipio=municipio,
        departamento="Cauca",
        periodo="2024-2027",
        es_municipio_pdet=es_pdet,
        extraction_timestamp=datetime.now(),
        extraction_method="unknown"
    )


# ============================================================================
# CONSTANTES DEL DOMINIO
# ============================================================================

# Municipios PDET de la subregión Alto Patía y Norte del Cauca
MUNICIPIOS_PDET_ALTO_PATIA_NORTE_CAUCA = [
    "Buenos Aires", "Caldono", "Caloto", "Corinto", "El Tambo",
    "Jambaló", "Mercaderes", "Miranda", "Morales", "Piendamó",
    "Santander de Quilichao", "Suárez", "Toribío"
]

# ODS relacionados con contexto PDET (Objetivos de Desarrollo Sostenible)
ODS_PDET_PRIORITARIOS = [1, 2, 3, 5, 8, 10, 11, 15, 16]
```

```
# 1-Fin pobreza, 2-Hambre cero, 3-Salud, 5-Igualdad género,
# 8-Trabajo decente, 10-Reducción desigualdades, 11-Ciudades sostenibles,
# 15-Vida ecosistemas terrestres, 16-Paz y justicia

# Sectores presupuestales estándar DNP (ejemplos)
SECTORES_DNP = {
    "12": "Justicia y del Derecho",
    "33": "Salud y Protección Social",
    "41": "Inclusión Social y Reconciliación",
    "43": "Agropecuario",
    "46": "Transporte",
    "47": "Vivienda, Ciudad y Territorio",
}

__version__ = "2.0.0"
__author__ = "F.A.R.F.A.N Policy Analysis Team"
```

src/farfan_pipeline/dashboard_atroz_/__init__.py

"""Dashboard / UI integration layer."""

src/farfan_pipeline/dashboard_atroz_/api_server.py

"""Shim to run the current dashboard Flask API server.

Keeps import paths stable by referencing the existing server implementation
under `farfan_pipeline.api.api_server`.
"""

from ..api.api_server import app

if __name__ == "__main__":
    # Delegate to the existing Flask app run configuration
    app.run(host="0.0.0.0", port=5000, debug=True)

src/farfan_pipeline/dashboard_atroz_/api_v1_errors.py

```
"""AtroZ Dashboard API error helpers."""

from __future__ import annotations

from dataclasses import dataclass
from typing import Any

from fastapi.responses import JSONResponse

from .api_v1_schemas import APIError


@dataclass(frozen=True, slots=True)
class AtrozAPIException(Exception):
    status: int
    code: str
    message: str
    details: dict[str, Any] | None = None
    retry_after: int | None = None


def api_error_response(exc: AtrozAPIException) -> JSONResponse:
    payload = APIError(
        status=exc.status,
        code=exc.code,
        message=exc.message,
        details=exc.details,
        retryAfter=exc.retry_after,
    ).model_dump(mode="json")

    headers: dict[str, str] = {}
    if exc.retry_after is not None:
        headers["Retry-After"] = str(exc.retry_after)

    return JSONResponse(status_code=exc.status, content=payload, headers=headers)
```

src/farfan_pipeline/dashboard_atroz_/api_v1_router.py

```
"""AtroZ Dashboard API v1 routes."""

from __future__ import annotations

import asyncio
import json
import os
from datetime import datetime, timezone
from typing import Any
from uuid import UUID

import structlog
from fastapi import APIRouter, Body, Depends, Query, Request, Response, WebSocket,
WebSocketDisconnect
from fastapi.responses import JSONResponse
from sse_starlette.sse import EventSourceResponse

from .api_v1_errors import AtrozAPIException, api_error_response
from .api_v1_schemas import (
    APIError,
    ClusterData,
    ComparisonMatrix,
    ComparisonMatrixRequest,
    ConstellationData,
    DashboardIngestRequest,
    ExportRequest,
    MunicipalAnalysis,
    Municipality,
    PDETRegion,
    QuestionAnalysis,
    TimelinePoint,
)
from .api_v1_store import AtrozStore

logger = structlog.get_logger(__name__)


class WebSocketHub:
    def __init__(self) -> None:
        self._connections: set[WebSocket] = set()
        self._lock = asyncio.Lock()

    async def connect(self, websocket: WebSocket) -> None:
        await websocket.accept()
        async with self._lock:
            self._connections.add(websocket)

    async def disconnect(self, websocket: WebSocket) -> None:
        async with self._lock:
            self._connections.discard(websocket)

    async def broadcast_json(self, payload: dict[str, Any]) -> None:
```

```
        async with self._lock:
            targets = list(self._connections)

        for ws in targets:
            try:
                await ws.send_json(payload)
            except Exception:
                await self.disconnect(ws)


class SSEHub:
    def __init__(self) -> None:
        self._queue: asyncio.Queue[dict[str, str]] = asyncio.Queue()

    async def publish(self, event: str, data: dict[str, Any]) -> None:
                    await   self._queue.put({"event":   event,   "data":   json.dumps(data,
ensure_ascii=False)})

    async def events(self, request: Request) -> Any:
        while True:
            if await request.is_disconnected():
                break
            try:
                event = await asyncio.wait_for(self._queue.get(), timeout=30.0)
                yield event
            except TimeoutError:
                yield {
                    "event": "heartbeat",
                                                        "data":   json.dumps({"timestamp":
datetime.now(timezone.utc).isoformat()}),
                }


STORE = AtrozStore()
WS_HUB = WebSocketHub()
SSE_HUB = SSEHub()

router = APIRouter(prefix="/api/v1", tags=["atroz-dashboard"])


def _api_error(status: int, code: str, message: str, details: dict[str, Any] | None =
None) -> JSONResponse:
                payload    =    APIError(status=status,    code=code,    message=message,
details=details).model_dump(mode="json")
    return JSONResponse(status_code=status, content=payload)


def _require_headers(request: Request) -> None:
    client = request.headers.get("X-Atroz-Client")
    version = request.headers.get("X-Atroz-Version")
    request_id = request.headers.get("X-Request-ID")

    if not client or not version or not request_id:
               raise  AtrozAPIException(status=400,  code="BAD_REQUEST",  message="Missing
```

```
required headers")

    if client != "dashboard-v1":
                raise  AtrozAPIException(status=400,  code="BAD_REQUEST",  message="Invalid
X-Atroz-Client")

    try:
        UUID(request_id)
    except ValueError as exc:
                raise  AtrozAPIException(status=400,  code="BAD_REQUEST",  message="Invalid
X-Request-ID") from exc

    if os.getenv("ATROZ_AUTH_REQUIRED", "false").lower() == "true":
        auth = request.headers.get("Authorization", "")
        if not auth.startswith("Bearer ") or len(auth) < 16:
                                raise   AtrozAPIException(status=401,   code="UNAUTHORIZED",
message="Missing/invalid Authorization")


async def require_atroz_headers(request: Request) -> None:
    _require_headers(request)


@router.post("/data/ingest")
async def ingest_data(
    payload: DashboardIngestRequest = Body(...),
    _: None = Depends(require_atroz_headers),
) -> Response:
    try:
        result = await STORE.ingest(payload)
    except ValueError as exc:
        return _api_error(400, "BAD_REQUEST", str(exc))
    except AtrozAPIException as exc:
        return api_error_response(exc)
    except Exception as exc:
        logger.exception("atroz_ingest_error", error=str(exc))
        return _api_error(500, "SERVER_ERROR", "Failed to ingest data")

    if result.get("mode") == "orchestrator":
        municipality_id = str(result.get("municipality_id", ""))
        region_id = str(result.get("region_id", ""))
        region = await STORE.get_region(region_id)
        municipality = await STORE.get_municipality(municipality_id)

        if region is not None:
            await WS_HUB.broadcast_json(
                {
                    "event": "DATA_UPDATED",
                    "type": "region",
                    "id": region_id,
                    "data": region.model_dump(mode="json"),
                    "timestamp": payload.timestamp.astimezone(timezone.utc).isoformat(),
                }
            )
```

```
        if municipality is not None:
            score_change = result.get("score_change") or {}
            await WS_HUB.broadcast_json(
                {
                    "event": "SCORE_CHANGED",
                    "entity": "municipality",
                    "id": municipality_id,
                    "newScore": score_change.get("new"),
                    "oldScore": score_change.get("old"),
                    "delta": score_change.get("delta"),
                }
            )

        await SSE_HUB.publish(
            "data.refresh",
            {
                "municipalityId": municipality_id,
                "regionId": region_id,
                "runId": payload.run_id,
                "timestamp": payload.timestamp.astimezone(timezone.utc).isoformat(),
            },
        )

    return JSONResponse(status_code=200, content=result)


@router.get("/pdet/regions", response_model=list[PDETRegion])
async def list_regions(response: Response, _: None = Depends(require_atroz_headers)) ->
list[PDETRegion]:
    response.headers["Cache-Control"] = "max-age=300"
    return await STORE.list_regions()


@router.get("/pdet/regions/{region_id}", response_model=PDETRegion)
async    def    get_region(region_id:    str,    response:    Response,    _:    None    =
Depends(require_atroz_headers)) -> Response:
    region = await STORE.get_region(region_id)
    if region is None:
        return _api_error(404, "NOT_FOUND", f"Region '{region_id}' not found")
    response.headers["Cache-Control"] = "max-age=300"
    return JSONResponse(status_code=200, content=region.model_dump(mode="json"))


@router.get("/pdet/regions/{region_id}/municipalities",
response_model=list[Municipality])
async def list_region_municipalities(
    region_id: str, response: Response, _: None = Depends(require_atroz_headers)
) -> list[Municipality]:
    response.headers["Cache-Control"] = "max-age=600"
    return await STORE.list_region_municipalities(region_id)


@router.get("/municipalities/{municipality_id}", response_model=Municipality)
```

```
async def get_municipality(
    municipality_id: str, response: Response, _: None = Depends(require_atroz_headers)
) -> Response:
    municipality = await STORE.get_municipality(municipality_id)
    if municipality is None:
            return _api_error(404, "NOT_FOUND", f"Municipality '{municipality_id}' not
found")
    response.headers["Cache-Control"] = "max-age=600"
    return JSONResponse(status_code=200, content=municipality.model_dump(mode="json"))


@router.get("/municipalities/{municipality_id}/analysis",
response_model=MunicipalAnalysis)
async def get_municipality_analysis(
    municipality_id: str, response: Response, _: None = Depends(require_atroz_headers)
) -> Response:
    analysis = await STORE.get_municipality_analysis(municipality_id)
    if analysis is None:
            return _api_error(404, "NOT_FOUND", f"Municipality '{municipality_id}' not
found")
    response.headers["Cache-Control"] = "max-age=900"
    return JSONResponse(status_code=200, content=analysis.model_dump(mode="json"))


@router.get("/analysis/clusters/{region_id}", response_model=list[ClusterData])
async def get_cluster_analysis(
    region_id: str, response: Response, _: None = Depends(require_atroz_headers)
) -> list[ClusterData]:
    response.headers["Cache-Control"] = "max-age=900"
    return await STORE.get_cluster_analysis(region_id)


@router.get("/analysis/questions/{municipality_id}",
response_model=list[QuestionAnalysis])
async def get_questions(
    municipality_id: str, response: Response, _: None = Depends(require_atroz_headers)
) -> list[QuestionAnalysis]:
    response.headers["Cache-Control"] = "max-age=900"
    return await STORE.get_questions(municipality_id)


@router.get("/visualization/constellation", response_model=ConstellationData)
async def get_constellation(response: Response, _: None =
Depends(require_atroz_headers)) -> ConstellationData:
    response.headers["Cache-Control"] = "max-age=1800"
    data = await STORE.build_constellation()
    return data


@router.get("/visualization/radar/{municipality_id}", response_model=dict[str, float])
async def get_radar(municipality_id: str, response: Response, _: None =
Depends(require_atroz_headers)) -> Response:
    analysis = await STORE.get_municipality_analysis(municipality_id)
    if analysis is None:
```

```
            return _api_error(404, "NOT_FOUND", f"Municipality '{municipality_id}' not
found")
    response.headers["Cache-Control"] = "max-age=900"
    return JSONResponse(status_code=200, content=analysis.radar)


@router.get("/visualization/phylogram/{region_id}")
async def get_phylogram(region_id: str, _: None = Depends(require_atroz_headers)) ->
dict[str, Any]:
    return {"regionId": region_id, "type": "phylogram", "data": []}


@router.get("/visualization/mesh/{region_id}")
async def get_mesh(region_id: str, _: None = Depends(require_atroz_headers)) ->
dict[str, Any]:
    return {"regionId": region_id, "type": "mesh", "data": []}


@router.get("/visualization/helix/{region_id}")
async def get_helix(region_id: str, _: None = Depends(require_atroz_headers)) ->
dict[str, Any]:
    return {"regionId": region_id, "type": "helix", "data": []}


@router.get("/timeline/regions/{region_id}", response_model=list[TimelinePoint])
async def get_timeline(region_id: str, _: None = Depends(require_atroz_headers)) ->
list[TimelinePoint]:
    return await STORE.get_timeline(region_id)


@router.get("/comparison/regions", response_model=ComparisonMatrix)
async def compare_regions(
    ids: str = Query(..., description="Comma-separated region IDs"),
    metrics: str | None = Query(default=None, description="Comma-separated metrics"),
    _: None = Depends(require_atroz_headers),
) -> ComparisonMatrix:
    id_list = [value.strip() for value in ids.split(",") if value.strip()]
        metric_list = [value.strip() for value in (metrics or "").split(",") if
value.strip()]
    return await STORE.compare("region", id_list, metric_list)


@router.post("/comparison/matrix", response_model=ComparisonMatrix)
async def compare_matrix(
    payload: ComparisonMatrixRequest, _: None = Depends(require_atroz_headers)
) -> ComparisonMatrix:
    return await STORE.compare(payload.entityType, payload.ids, payload.metrics)


@router.get("/evidence/stream")
async def evidence_stream(request: Request, _: None = Depends(require_atroz_headers)) ->
EventSourceResponse:
    return EventSourceResponse(SSE_HUB.events(request))
```

```
@router.get("/documents/references")
async def get_references(_: None = Depends(require_atroz_headers)) -> dict[str, Any]:
    return {"references": []}


@router.get("/documents/sources")
async def get_sources(_: None = Depends(require_atroz_headers)) -> dict[str, Any]:
    return {"sources": []}


@router.get("/documents/citations")
async def get_citations(_: None = Depends(require_atroz_headers)) -> dict[str, Any]:
    return {"citations": []}


@router.post("/export/dashboard")
async def export_dashboard(
    payload: ExportRequest, _: None = Depends(require_atroz_headers)
) -> Response:
    return _export_bytes("dashboard", payload)


@router.post("/export/region")
async      def      export_region(payload:      ExportRequest,      _:      None      =
Depends(require_atroz_headers)) -> Response:
    return _export_bytes("region", payload)


@router.post("/export/comparison")
async     def     export_comparison(payload:     ExportRequest,     _:     None     =
Depends(require_atroz_headers)) -> Response:
    return _export_bytes("comparison", payload)


@router.post("/export/reports")
async      def      export_reports(payload:      ExportRequest,      _:      None      =
Depends(require_atroz_headers)) -> Response:
    return _export_bytes("reports", payload)


@router.post("/export/municipality")
async def export_municipality(
    payload: ExportRequest, _: None = Depends(require_atroz_headers)
) -> Response:
    return _export_bytes("municipality", payload)


@router.websocket("/realtime")
async def realtime(websocket: WebSocket) -> None:
    await WS_HUB.connect(websocket)
    try:
        while True:
            await websocket.receive_text()
```

```
    except WebSocketDisconnect:
        await WS_HUB.disconnect(websocket)


def _export_bytes(scope: str, payload: ExportRequest) -> Response:
    if payload.format == "png":
        data = _transparent_png_1x1()
        return Response(
            content=data,
            media_type="image/png",
            headers={"Content-Disposition": f"attachment; filename=\"{scope}.png\""},
        )
    if payload.format == "svg":
                    svg   =   "<svg   xmlns=\"http://www.w3.org/2000/svg\"   width=\"1\"
height=\"1\"></svg>"
        return Response(
            content=svg.encode("utf-8"),
            media_type="image/svg+xml",
            headers={"Content-Disposition": f"attachment; filename=\"{scope}.svg\""},
        )
    if payload.format == "pdf":
        pdf = _minimal_pdf()
        return Response(
            content=pdf,
            media_type="application/pdf",
            headers={"Content-Disposition": f"attachment; filename=\"{scope}.pdf\""},
        )
    html = "<!doctype html><html><body><pre>Export placeholder</pre></body></html>"
    return Response(
        content=html.encode("utf-8"),
        media_type="text/html",
        headers={"Content-Disposition": f"attachment; filename=\"{scope}.html\""},
    )


def _transparent_png_1x1() -> bytes:
    return bytes.fromhex(
        "89504E470D0A1A0A0000000D4948445200000001000000010806000000 1F15C489"
        "0000000A49444154789C6300010000050001 0D0A2DB40000000049454E44AE426082"
    )


def _minimal_pdf() -> bytes:
    return (
        b"%PDF-1.4\n1 0 obj<<>>endobj\n"
        b"2 0 obj<< /Type /Catalog /Pages 3 0 R >>endobj\n"
        b"3 0 obj<< /Type /Pages /Kids [4 0 R] /Count 1 >>endobj\n"
        b"4 0 obj<< /Type /Page /Parent 3 0 R /MediaBox [0 0 1 1] >>endobj\n"
        b"xref\n0 5\n0000000000 65535 f \n0000000009 00000 n \n0000000030 00000 n \n"
        b"0000000079 00000 n \n0000000128 00000 n \ntrailer<< /Size 5 /Root 2 0 R >>\n"
        b"startxref\n190\n%%EOF\n"
    )
```

```
src/farfan_pipeline/dashboard_atroz_/api_v1_schemas.py

"""AtroZ Dashboard API v1 schemas.

These models define the wire contracts for the dashboard backend. They are intentionally
strict (extra fields forbidden) for outward-facing responses, while ingest payloads
allow
extra fields to preserve full orchestrator artifacts.
"""

from __future__ import annotations

from datetime import datetime
from typing import Any, Literal

from pydantic import BaseModel, ConfigDict, Field


class APIError(BaseModel):
    model_config = ConfigDict(extra="forbid")

    status: int
    code: str
    message: str
    details: dict[str, Any] | None = None
    retryAfter: int | None = None


class Coordinates2D(BaseModel):
    model_config = ConfigDict(extra="forbid")

    x: float
    y: float


class PDETRegionScores(BaseModel):
    model_config = ConfigDict(extra="forbid")

    overall: float
    governance: float
    social: float
    economic: float
    environmental: float


class PDETRegion(BaseModel):
    model_config = ConfigDict(extra="forbid")

    id: str
    name: str
    municipalities: int
    scores: PDETRegionScores
    coordinates: Coordinates2D
```

```
class ClusterData(BaseModel):
    model_config = ConfigDict(extra="forbid")

    id: str
    score: float
    normalized_score: float


class QuestionAnalysis(BaseModel):
    model_config = ConfigDict(extra="forbid")

    questionId: int = Field(..., ge=0)
    score: float
    evidence: list[str]
    quality: Literal["EXCELENTE", "ACEPTABLE", "INSUFICIENTE", "NO_APLICABLE", "ERROR"]


class MunicipalAnalysis(BaseModel):
    model_config = ConfigDict(extra="forbid")

    radar: dict[str, float]
    clusters: list[ClusterData]
    questions: list[QuestionAnalysis]
    recommendations: list[str] = Field(default_factory=list)


class Municipality(BaseModel):
    model_config = ConfigDict(extra="forbid")

    id: str
    name: str
    department: str
    regionId: str
    population: int = Field(default=0, ge=0)
    analysis: MunicipalAnalysis | None = None


class ConstellationNodePosition(BaseModel):
    model_config = ConfigDict(extra="forbid")

    x: float
    y: float


class ConstellationNodeProperties(BaseModel):
    model_config = ConfigDict(extra="forbid")

    size: float
    color: str
    pulseRate: float
    connectionStrength: float
```

```
class ConstellationNode(BaseModel):
    model_config = ConfigDict(extra="forbid")

    id: str
    position: ConstellationNodePosition
    properties: ConstellationNodeProperties


class ConstellationConnection(BaseModel):
    model_config = ConfigDict(extra="forbid")

    source: str
    target: str
    strength: float
    type: Literal["geographic", "score_similarity", "pdet_region"]


class ConstellationData(BaseModel):
    model_config = ConfigDict(extra="forbid")

    nodes: list[ConstellationNode]
    connections: list[ConstellationConnection]


class TimelinePoint(BaseModel):
    model_config = ConfigDict(extra="forbid")

    timestamp: str
    scores: dict[str, float]
    events: list[str]


class ComparisonMatrix(BaseModel):
    model_config = ConfigDict(extra="forbid")

    entityType: Literal["region", "municipality"]
    ids: list[str]
    metrics: list[str]
    matrix: dict[str, dict[str, float]]


class ComparisonMatrixRequest(BaseModel):
    model_config = ConfigDict(extra="forbid")

    entityType: Literal["region", "municipality"]
    ids: list[str] = Field(..., min_length=1, max_length=20)
    metrics: list[str] = Field(default_factory=list)


class ExportRequest(BaseModel):
    model_config = ConfigDict(extra="forbid")

    format: Literal["png", "svg", "pdf", "html"]
    resolution: Literal["720p", "1080p", "4k"] = "1080p"
```

```
    includeData: bool = True


class MunicipalitySelector(BaseModel):
    model_config = ConfigDict(extra="forbid")

    id: str | None = None
    dane_code: str | None = None
    name: str | None = None
    department: str | None = None
    document_id: str | None = None
    pdf_path: str | None = None


class MacroClusterScoreDataIn(BaseModel):
    model_config = ConfigDict(extra="allow")

    id: str
    score: float
    normalized_score: float


class MacroEvaluationIn(BaseModel):
    model_config = ConfigDict(extra="allow")

    macro_score: float
    macro_score_normalized: float
    clusters: list[MacroClusterScoreDataIn]
    details: dict[str, Any] | None = None


class ClusterScoreIn(BaseModel):
    model_config = ConfigDict(extra="allow")

    cluster_id: str
    cluster_name: str | None = None
    score: float
    coherence: float | None = None


class ScoredMicroQuestionIn(BaseModel):
    model_config = ConfigDict(extra="allow")

    question_id: str
    question_global: int
    score: float | None = None
    normalized_score: float | None = None
    quality_level: str | None = None
    evidence: dict[str, Any] | None = None


class DashboardIngestRequest(BaseModel):
    model_config = ConfigDict(extra="forbid")
```

```
run_id: str
timestamp: datetime
municipality: MunicipalitySelector | None = None

macro_result: MacroEvaluationIn | None = None
cluster_scores: list[ClusterScoreIn] | None = None
policy_area_scores: list[dict[str, Any]] | None = None
dimension_scores: list[dict[str, Any]] | None = None
scored_results: list[ScoredMicroQuestionIn] | None = None
micro_results: list[dict[str, Any]] | None = None

pdet_regions: list[PDETRegion] | None = None
municipalities: list[Municipality] | None = None
```

```
src/farfan_pipeline/dashboard_atroz_/api_v1_store.py

"""In-memory AtroZ dashboard store.

The AtroZ dashboard needs a complete baseline of 170 PDET municipalities and 16 subregions
even before any analysis runs complete. This store bootstraps reference data from
`pdet_colombia_data` and applies incremental updates via the ingest endpoint.
"""

from __future__ import annotations

import asyncio
import hashlib
from collections import deque
from dataclasses import dataclass, field
from datetime import datetime, timezone
from typing import Any, Literal

from .api_v1_schemas import (
    ClusterData,
    ComparisonMatrix,
    ConstellationConnection,
    ConstellationData,
    ConstellationNode,
    ConstellationNodePosition,
    ConstellationNodeProperties,
    Coordinates2D,
    DashboardIngestRequest,
    MunicipalAnalysis,
    Municipality,
    MunicipalitySelector,
    PDETRegion,
    PDETRegionScores,
    QuestionAnalysis,
    TimelinePoint,
)
from .api_v1_utils import slugify
from .pdet_colombia_data import PDETSubregion, PDET_MUNICIPALITIES, PDETMunicipality


ExpectedClusterKey = Literal["governance", "social", "economic", "environmental"]


@dataclass(frozen=True, slots=True)
class RegionDefinition:
    id: str
    name: str
    coordinates: Coordinates2D


REGION_DEFINITIONS: dict[PDETSubregion, RegionDefinition] = {
    PDETSubregion.ALTO_PATIA: RegionDefinition(
        id="alto-patia",
```

```
    name="Alto Patía y Norte del Cauca",
    coordinates=Coordinates2D(x=25.0, y=20.0),
),
PDETSubregion.ARAUCA: RegionDefinition(
    id="arauca",
    name="Arauca",
    coordinates=Coordinates2D(x=75.0, y=15.0),
),
PDETSubregion.BAJO_CAUCA: RegionDefinition(
    id="bajo-cauca",
    name="Bajo Cauca y Nordeste Antioqueño",
    coordinates=Coordinates2D(x=45.0, y=25.0),
),
PDETSubregion.CATATUMBO: RegionDefinition(
    id="catatumbo",
    name="Catatumbo",
    coordinates=Coordinates2D(x=65.0, y=20.0),
),
PDETSubregion.CHOCO: RegionDefinition(
    id="choco",
    name="Chocó",
    coordinates=Coordinates2D(x=15.0, y=35.0),
),
PDETSubregion.CAGUAN: RegionDefinition(
    id="caguan",
    name="Cuenca del Caguán y Piedemonte Caqueteño",
    coordinates=Coordinates2D(x=55.0, y=40.0),
),
PDETSubregion.MACARENA: RegionDefinition(
    id="macarena",
    name="Macarena-Guaviare",
    coordinates=Coordinates2D(x=60.0, y=55.0),
),
PDETSubregion.MONTES_MARIA: RegionDefinition(
    id="montes-maria",
    name="Montes de María",
    coordinates=Coordinates2D(x=40.0, y=10.0),
),
PDETSubregion.PACIFICO_MEDIO: RegionDefinition(
    id="pacifico-medio",
    name="Pacífico Medio",
    coordinates=Coordinates2D(x=10.0, y=50.0),
),
PDETSubregion.PACIFICO_NARINENSE: RegionDefinition(
    id="pacifico-narinense",
    name="Pacífico y Frontera Nariñense",
    coordinates=Coordinates2D(x=5.0, y=65.0),
),
PDETSubregion.PUTUMAYO: RegionDefinition(
    id="putumayo",
    name="Putumayo",
    coordinates=Coordinates2D(x=35.0, y=70.0),
),
PDETSubregion.SIERRA_NEVADA: RegionDefinition(
```

```
        id="sierra-nevada",
        name="Sierra Nevada - Perijá - Zona Bananera",
        coordinates=Coordinates2D(x=70.0, y=5.0),
    ),
    PDETSubregion.SUR_BOLIVAR: RegionDefinition(
        id="sur-bolivar",
        name="Sur de Bolívar",
        coordinates=Coordinates2D(x=50.0, y=15.0),
    ),
    PDETSubregion.SUR_CORDOBA: RegionDefinition(
        id="sur-cordoba",
        name="Sur de Córdoba",
        coordinates=Coordinates2D(x=35.0, y=15.0),
    ),
    PDETSubregion.SUR_TOLIMA: RegionDefinition(
        id="sur-tolima",
        name="Sur del Tolima",
        coordinates=Coordinates2D(x=45.0, y=45.0),
    ),
    PDETSubregion.URABA: RegionDefinition(
        id="uraba",
        name="Urabá Antioqueño",
        coordinates=Coordinates2D(x=20.0, y=10.0),
    ),
}


def municipality_id_for(municipality: PDETMunicipality) -> str:
    return f"{slugify(municipality.name)}-{slugify(municipality.department)}"


def municipality_id_for_parts(name: str, department: str) -> str:
    return f"{slugify(name)}-{slugify(department)}"


def _cluster_key_from_text(value: str) -> ExpectedClusterKey | None:
    slug = slugify(value)
    if "gobern" in slug:
        return "governance"
    if "social" in slug:
        return "social"
    if "econ" in slug:
        return "economic"
    if "ambient" in slug:
        return "environmental"
    return None


def    normalize_cluster_key(cluster_id:    str,    cluster_name:    str    |    None)    ->
ExpectedClusterKey | None:
                    return     _cluster_key_from_text(cluster_name     or     "")     or
_cluster_key_from_text(cluster_id)
```

```
def score_0_to_3_to_percent(score: float) -> float:
    return (score / 3.0) * 100.0


def score_color(score_percent: float | None) -> str:
    if score_percent is None:
        return "#666666"
    if score_percent < 40.0:
        return "#C41E3A"
    if score_percent < 70.0:
        return "#B2642E"
    return "#39FF14"


@dataclass(slots=True)
class MunicipalityState:
    id: str
    name: str
    department: str
    region_id: str
    dane_code: str
    population: int
    latitude: float
    longitude: float
    last_updated: datetime | None = None
    latest_run_id: str | None = None
    overall_score_percent: float | None = None
                    cluster_scores_percent:     dict[ExpectedClusterKey,     float]     =
field(default_factory=dict)
    question_scores: dict[int, QuestionAnalysis] = field(default_factory=dict)
    radar: dict[str, float] = field(default_factory=dict)

    def to_api(self) -> Municipality:
        analysis: MunicipalAnalysis | None = None
        if self.overall_score_percent is not None:
            analysis = MunicipalAnalysis(
                radar=dict(self.radar),
                clusters=[
                    ClusterData(
                        id=key,
                        score=self.cluster_scores_percent.get(key, 0.0),
                             normalized_score=self.cluster_scores_percent.get(key, 0.0) /
100.0,
                    )
                    for key in ("governance", "social", "economic", "environmental")
                ],
                questions=list(self.question_scores.values()),
                recommendations=[],
            )

        return Municipality(
            id=self.id,
            name=self.name,
            department=self.department,
```

```
            regionId=self.region_id,
            population=self.population,
            analysis=analysis,
        )


@dataclass(slots=True)
class RegionState:
    id: str
    name: str
    coordinates: Coordinates2D
    municipality_ids: list[str]
    scores: PDETRegionScores = field(
        default_factory=lambda: PDETRegionScores(
            overall=0.0,
            governance=0.0,
            social=0.0,
            economic=0.0,
            environmental=0.0,
        )
    )

    def to_api(self) -> PDETRegion:
        return PDETRegion(
            id=self.id,
            name=self.name,
            municipalities=len(self.municipality_ids),
            scores=self.scores,
            coordinates=self.coordinates,
        )


class AtrozStore:
    def __init__(self) -> None:
        self._lock = asyncio.Lock()
        self._municipalities: dict[str, MunicipalityState] = {}
        self._regions: dict[str, RegionState] = {}
        self._timeline: dict[str, deque[TimelinePoint]] = {}

        self._bootstrap_reference_data()

    def _bootstrap_reference_data(self) -> None:
        if len(REGION_DEFINITIONS) != 16:
              raise RuntimeError(f"PDET subregion definitions mismatch: expected 16, got
{len(REGION_DEFINITIONS)}")
        if len(PDET_MUNICIPALITIES) != 170:
                  raise RuntimeError(f"PDET municipalities mismatch: expected 170, got
{len(PDET_MUNICIPALITIES)}")

              by_region: dict[str, list[str]] = {definition.id: [] for definition in
REGION_DEFINITIONS.values()}

        for municipality in PDET_MUNICIPALITIES:
            region_def = REGION_DEFINITIONS.get(municipality.subregion)
```

```
            if region_def is None:
                continue
            mun_id = municipality_id_for(municipality)
            state = MunicipalityState(
                id=mun_id,
                name=municipality.name,
                department=municipality.department,
                region_id=region_def.id,
                dane_code=municipality.dane_code,
                population=int(municipality.population or 0),
                latitude=float(municipality.latitude or 0.0),
                longitude=float(municipality.longitude or 0.0),
            )
            self._municipalities[mun_id] = state
            by_region[region_def.id].append(mun_id)

        if len(self._municipalities) != 170:
            raise RuntimeError(
                        f"PDET municipality registry mismatch: expected 170, got
{len(self._municipalities)}"
            )

        for subregion, definition in REGION_DEFINITIONS.items():
            region_id = definition.id
            municipality_ids = sorted(by_region.get(region_id, []))
            self._regions[region_id] = RegionState(
                id=region_id,
                name=definition.name,
                coordinates=definition.coordinates,
                municipality_ids=municipality_ids,
            )
            self._timeline[region_id] = deque(maxlen=2048)

    async def list_regions(self) -> list[PDETRegion]:
        async with self._lock:
            return [region.to_api() for region in self._regions.values()]

    async def get_region(self, region_id: str) -> PDETRegion | None:
        async with self._lock:
            region = self._regions.get(region_id)
            return None if region is None else region.to_api()

    async def list_region_municipalities(self, region_id: str) -> list[Municipality]:
        async with self._lock:
            region = self._regions.get(region_id)
            if region is None:
                return []
                        return [self._municipalities[mun_id].to_api() for mun_id in
region.municipality_ids]

    async def get_municipality(self, municipality_id: str) -> Municipality | None:
        async with self._lock:
            state = self._municipalities.get(municipality_id)
            return None if state is None else state.to_api()
```

```
    async def get_municipality_analysis(self, municipality_id: str) -> MunicipalAnalysis
| None:
        async with self._lock:
            state = self._municipalities.get(municipality_id)
            if state is None:
                return None
            api_obj = state.to_api()
            return api_obj.analysis

    async def get_questions(self, municipality_id: str) -> list[QuestionAnalysis]:
        async with self._lock:
            state = self._municipalities.get(municipality_id)
            if state is None:
                return []
            return list(state.question_scores.values())

    async def get_cluster_analysis(self, region_id: str) -> list[ClusterData]:
        async with self._lock:
            region = self._regions.get(region_id)
            if region is None:
                return []
            totals: dict[ExpectedClusterKey, float] = {
                "governance": 0.0,
                "social": 0.0,
                "economic": 0.0,
                "environmental": 0.0,
            }
            count = 0
            for mun_id in region.municipality_ids:
                mun = self._municipalities[mun_id]
                if mun.overall_score_percent is None:
                    continue
                count += 1
                for key in totals:
                    totals[key] += mun.cluster_scores_percent.get(key, 0.0)

            denom = float(count) if count > 0 else 1.0
            return [
                                        ClusterData(id=key,  score=totals[key]  /  denom,
normalized_score=(totals[key] / denom) / 100.0)
                for key in ("governance", "social", "economic", "environmental")
            ]

    async def get_timeline(self, region_id: str) -> list[TimelinePoint]:
        async with self._lock:
            points = self._timeline.get(region_id)
            if points is None:
                return []
            return list(points)

    async def ingest(self, payload: DashboardIngestRequest) -> dict[str, Any]:
        if payload.municipalities is not None and payload.pdet_regions is not None:
            return await self._ingest_snapshot(payload)
```

```
        return await self._ingest_orchestrator_update(payload)

    async def _ingest_snapshot(self, payload: DashboardIngestRequest) -> dict[str, Any]:
        async with self._lock:
            if payload.municipalities is None or payload.pdet_regions is None:
                        raise ValueError("Snapshot ingest requires municipalities and
pdet_regions")

            incoming_regions = {region.id: region for region in payload.pdet_regions}
            for region_id, region_state in self._regions.items():
                if region_id in incoming_regions:
                    region_state.scores = incoming_regions[region_id].scores

            for municipality in payload.municipalities:
                existing = self._municipalities.get(municipality.id)
                if existing is None:
                    continue
                existing.population = municipality.population
                existing.last_updated = payload.timestamp
                existing.latest_run_id = payload.run_id
                if municipality.analysis and municipality.analysis.clusters:
                                                        existing.overall_score_percent   =
municipality.analysis.radar.get("overall")
                                existing.radar  =  dict(municipality.analysis.radar)  if
municipality.analysis else {}

            return {"status": "success", "mode": "snapshot"}

    def _resolve_municipality_by_dane(self, dane_code: str) -> MunicipalityState | None:
        for state in self._municipalities.values():
            if state.dane_code == dane_code:
                return state
        return None

       def  _resolve_municipality_by_name(self,  name:  str,  department:  str  |  None)  ->
MunicipalityState | None:
        if department:
            candidate_id = municipality_id_for_parts(name, department)
            return self._municipalities.get(candidate_id)

        slug_name = slugify(name)
                  matches  =  [state  for  state  in  self._municipalities.values()  if
slugify(state.name) == slug_name]
        if len(matches) == 1:
            return matches[0]
        return None

    def _resolve_municipality_from_selector(
        self, selector: MunicipalitySelector | None
    ) -> MunicipalityState | None:
        if selector is None:
            return None
        if selector.id:
            return self._municipalities.get(selector.id)
```

```
        if selector.dane_code:
            state = self._resolve_municipality_by_dane(selector.dane_code)
            if state is not None:
                return state
        if selector.name:
                            state  =  self._resolve_municipality_by_name(selector.name,
selector.department)
            if state is not None:
                return state
        return None

       async def _ingest_orchestrator_update(self, payload: DashboardIngestRequest) ->
dict[str, Any]:
        if payload.macro_result is None:
            raise ValueError("macro_result is required for orchestrator ingest")
        if payload.cluster_scores is None:
            raise ValueError("cluster_scores is required for orchestrator ingest")
        if payload.scored_results is None:
            raise ValueError("scored_results is required for orchestrator ingest")

        async with self._lock:
                                                                    municipality     =
self._resolve_municipality_from_selector(payload.municipality)
            if municipality is None:
                raise ValueError("Unable to resolve municipality for ingest")

            previous_score = municipality.overall_score_percent
            municipality.latest_run_id = payload.run_id
            municipality.last_updated = payload.timestamp

                                                  municipality.overall_score_percent    =
score_0_to_3_to_percent(payload.macro_result.macro_score)

            cluster_scores: dict[ExpectedClusterKey, float] = {}
            for cs in payload.cluster_scores:
                key = normalize_cluster_key(cs.cluster_id, cs.cluster_name)
                if key is None:
                    continue
                cluster_scores[key] = score_0_to_3_to_percent(cs.score)

            municipality.cluster_scores_percent = cluster_scores
            municipality.radar = {
                "overall": municipality.overall_score_percent,
                "governance": cluster_scores.get("governance", 0.0),
                "social": cluster_scores.get("social", 0.0),
                "economic": cluster_scores.get("economic", 0.0),
                "environmental": cluster_scores.get("environmental", 0.0),
                "infrastructure": 0.0,
                "security": 0.0,
            }

            municipality.question_scores = {}
            for entry in payload.scored_results:
                score_value = entry.normalized_score
```

```
                if score_value is None and entry.score is not None:
                    score_value = entry.score / 3.0
                if score_value is None:
                    continue
                municipality.question_scores[entry.question_global] = QuestionAnalysis(
                    questionId=entry.question_global,
                    score=float(score_value) * 100.0,
                    evidence=[],
                    quality=_quality_map(entry.quality_level),
                )

            self._recompute_region_scores_locked(municipality.region_id)
            self._append_timeline_locked(municipality.region_id, payload.timestamp)

                            score_delta = None if previous_score is None else
municipality.overall_score_percent - previous_score

            return {
                "status": "success",
                "mode": "orchestrator",
                "municipality_id": municipality.id,
                "region_id": municipality.region_id,
                "score_change": {
                    "old": previous_score,
                    "new": municipality.overall_score_percent,
                    "delta": score_delta,
                },
            }

    def _recompute_region_scores_locked(self, region_id: str) -> None:
        region = self._regions.get(region_id)
        if region is None:
            return

        totals: dict[ExpectedClusterKey, float] = {
            "governance": 0.0,
            "social": 0.0,
            "economic": 0.0,
            "environmental": 0.0,
        }
        overall_total = 0.0
        counted = 0

        for mun_id in region.municipality_ids:
            mun = self._municipalities[mun_id]
            if mun.overall_score_percent is None:
                continue
            counted += 1
            overall_total += mun.overall_score_percent
            for key in totals:
                totals[key] += mun.cluster_scores_percent.get(key, 0.0)

        denom = float(counted) if counted > 0 else 1.0
        region.scores = PDETRegionScores(
```

```
            overall=overall_total / denom,
            governance=totals["governance"] / denom,
            social=totals["social"] / denom,
            economic=totals["economic"] / denom,
            environmental=totals["environmental"] / denom,
        )

    def _append_timeline_locked(self, region_id: str, timestamp: datetime) -> None:
        points = self._timeline.get(region_id)
        region = self._regions.get(region_id)
        if points is None or region is None:
            return

        points.append(
            TimelinePoint(
                timestamp=timestamp.astimezone(timezone.utc).isoformat(),
                scores={
                    "overall": region.scores.overall,
                    "governance": region.scores.governance,
                    "social": region.scores.social,
                    "economic": region.scores.economic,
                    "environmental": region.scores.environmental,
                },
                events=["data.ingest"],
            )
        )

    async def build_constellation(self) -> ConstellationData:
        async with self._lock:
            nodes: list[ConstellationNode] = []
            connections: list[ConstellationConnection] = []

            for region in self._regions.values():
                nodes.append(
                    ConstellationNode(
                        id=region.id,
                               position=ConstellationNodePosition(x=region.coordinates.x,
y=region.coordinates.y),
                        properties=ConstellationNodeProperties(
                            size=180.0,
                            color="#00D4FF",
                            pulseRate=0.5,
                            connectionStrength=1.0,
                        ),
                    )
                )

                  min_pop = min((m.population for m in self._municipalities.values()),
default=0)
                  max_pop = max((m.population for m in self._municipalities.values()),
default=1)
            pop_span = max(max_pop - min_pop, 1)

            for municipality in self._municipalities.values():
```

```
                base = self._regions.get(municipality.region_id)
                if base is None:
                    continue
                jitter_x, jitter_y = _stable_jitter(municipality.id)
                x = base.coordinates.x + jitter_x
                y = base.coordinates.y + jitter_y

                size = 80.0 + 70.0 * ((municipality.population - min_pop) / pop_span)
                color = score_color(municipality.overall_score_percent)

                nodes.append(
                    ConstellationNode(
                        id=municipality.id,
                        position=ConstellationNodePosition(x=x, y=y),
                        properties=ConstellationNodeProperties(
                            size=size,
                            color=color,
                              pulseRate=0.5 + (municipality.overall_score_percent or 0.0)
/ 200.0,
                                connectionStrength=(municipality.overall_score_percent or
0.0) / 100.0,
                        ),
                    )
                )

                connections.append(
                    ConstellationConnection(
                        source=municipality.region_id,
                        target=municipality.id,
                        strength=1.0,
                        type="pdet_region",
                    )
                )

            _add_similarity_connections(nodes, connections, self._municipalities)

            return ConstellationData(nodes=nodes, connections=connections)

    async def compare(
        self,
        entity_type: Literal["region", "municipality"],
        ids: list[str],
        metrics: list[str],
    ) -> ComparisonMatrix:
        async with self._lock:
            matrix: dict[str, dict[str, float]] = {}
            if entity_type == "region":
                for region_id in ids:
                    region = self._regions.get(region_id)
                    if region is None:
                        continue
                    matrix[region_id] = _extract_metrics(region.scores, metrics)
            else:
                for mun_id in ids:
```

```
                    mun = self._municipalities.get(mun_id)
                    if mun is None:
                        continue
                    matrix[mun_id] = _extract_metrics_from_municipality(mun, metrics)

                return ComparisonMatrix(entityType=entity_type, ids=ids, metrics=metrics,
matrix=matrix)


def _quality_map(value: str | None) -> str:
    if value is None:
        return "NO_APLICABLE"
    slug = slugify(value)
    if "excel" in slug or slug in {"alta", "high"}:
        return "EXCELENTE"
    if "acept" in slug or slug in {"media", "medium"}:
        return "ACEPTABLE"
    if "insuf" in slug or slug in {"baja", "low"}:
        return "INSUFICIENTE"
    return "ERROR"


def _stable_jitter(identifier: str) -> tuple[float, float]:
    digest = hashlib.sha256(identifier.encode("utf-8")).digest()
    x_raw = int.from_bytes(digest[:2], "big") / 65535.0
    y_raw = int.from_bytes(digest[2:4], "big") / 65535.0
    return (x_raw - 0.5) * 8.0, (y_raw - 0.5) * 8.0


def _add_similarity_connections(
    nodes: list[ConstellationNode],
    connections: list[ConstellationConnection],
    municipalities: dict[str, MunicipalityState],
) -> None:
    scored = [m for m in municipalities.values() if m.overall_score_percent is not None]
    scored.sort(key=lambda m: m.overall_score_percent or 0.0)

    for i, mun in enumerate(scored):
        for j in (i - 1, i + 1):
            if j < 0 or j >= len(scored):
                continue
            other = scored[j]
             diff = abs((mun.overall_score_percent or 0.0) - (other.overall_score_percent
or 0.0))
            if diff > 5.0:
                continue
            strength = max(0.1, 1.0 - (diff / 5.0))
            connections.append(
                ConstellationConnection(
                    source=mun.id,
                    target=other.id,
                    strength=strength,
                    type="score_similarity",
                )
```

```
            )


def _extract_metrics(scores: PDETRegionScores, metrics: list[str]) -> dict[str, float]:
    if not metrics:
        return {
            "overall": scores.overall,
            "governance": scores.governance,
            "social": scores.social,
            "economic": scores.economic,
            "environmental": scores.environmental,
        }
    out: dict[str, float] = {}
    for metric in metrics:
        if hasattr(scores, metric):
            out[metric] = float(getattr(scores, metric))
    return out


def _extract_metrics_from_municipality(mun: MunicipalityState, metrics: list[str]) ->
dict[str, float]:
    if not metrics:
        return dict(mun.radar)
    out: dict[str, float] = {}
    for metric in metrics:
        value = mun.radar.get(metric)
        if value is not None:
            out[metric] = float(value)
    return out
```

src/farfan_pipeline/dashboard_atroz_/api_v1_utils.py

```
"""AtroZ Dashboard API v1 helpers."""

from __future__ import annotations

import re
import unicodedata


_NON_ALNUM_RE = re.compile(r"[^a-z0-9]+")


def slugify(value: str) -> str:
    normalized = unicodedata.normalize("NFKD", value)
    ascii_value = normalized.encode("ascii", "ignore").decode("ascii")
    lowered = ascii_value.lower()
    slug = _NON_ALNUM_RE.sub("-", lowered).strip("-")
    return slug
```

src/farfan_pipeline/dashboard_atroz_/auth_admin.py

```
"""
AtroZ Admin Authentication Module
Minimal but secure authentication for admin panel access
"""

import hashlib
import logging
import secrets
import time
from dataclasses import dataclass
from datetime import datetime, timedelta

logger = logging.getLogger(__name__)


@dataclass
class AdminSession:
    """Represents an active admin session"""
    session_id: str
    username: str
    created_at: datetime
    last_activity: datetime
    ip_address: str

    def is_expired(self, timeout_minutes: int = 60) -> bool:
        """Check if session has expired"""
        return datetime.now() - self.last_activity > timedelta(minutes=timeout_minutes)

    def update_activity(self) -> None:
        """Update last activity timestamp"""
        self.last_activity = datetime.now()


class AdminAuthenticator:
    """
    Simple but secure authentication system for admin panel.

    Security features:
    - Password hashing with salt
    - Session management with timeout
    - Rate limiting on login attempts
    - IP-based session tracking
    """

    def __init__(self, session_timeout_minutes: int = 60) -> None:
        self.session_timeout = session_timeout_minutes
        self.sessions: dict[str, AdminSession] = {}
        self.login_attempts: dict[str, list] = {}

        # Default credentials (should be changed in production)
        # Default password: "atroz_admin_2024"
        self.users = {
```

```
            "admin": {
                                "password_hash":  self._hash_password("atroz_admin_2024",
"default_salt"),
                "salt": "default_salt",
                "role": "administrator"
            }
        }

        logger.info("Admin authenticator initialized")

    def _hash_password(self, password: str, salt: str) -> str:
        """Hash password with salt using SHA-256"""
        return hashlib.sha256(f"{password}{salt}".encode()).hexdigest()

    def _generate_session_id(self) -> str:
        """Generate secure random session ID"""
        return secrets.token_urlsafe(32)

     def _check_rate_limit(self, ip_address: str, max_attempts: int = 5, window_minutes:
int = 15) -> bool:
        """Check if IP has exceeded login attempt rate limit"""
        now = time.time()
        window_seconds = window_minutes * 60

        if ip_address not in self.login_attempts:
            self.login_attempts[ip_address] = []

        # Remove old attempts outside window
        self.login_attempts[ip_address] = [
            timestamp for timestamp in self.login_attempts[ip_address]
            if now - timestamp < window_seconds
        ]

        # Check if too many attempts
        if len(self.login_attempts[ip_address]) >= max_attempts:
            logger.warning(f"Rate limit exceeded for IP: {ip_address}")
            return False

        return True

    def authenticate(self, username: str, password: str, ip_address: str) -> str | None:
        """
        Authenticate user and create session.

        Args:
            username: Username to authenticate
            password: Plain text password
            ip_address: IP address of client

        Returns:
            Session ID if authentication successful, None otherwise
        """
        # Check rate limit
        if not self._check_rate_limit(ip_address):
```

```
            return None

        # Record login attempt
        if ip_address in self.login_attempts:
            self.login_attempts[ip_address].append(time.time())
        else:
            self.login_attempts[ip_address] = [time.time()]

        # Check if user exists
        if username not in self.users:
            logger.warning(f"Login attempt for non-existent user: {username}")
            return None

        user = self.users[username]
        password_hash = self._hash_password(password, user["salt"])

        # Verify password
        if password_hash != user["password_hash"]:
                  logger.warning(f"Failed  login  attempt  for  user:  {username}  from  IP:
{ip_address}")
            return None

        # Create session
        session_id = self._generate_session_id()
        self.sessions[session_id] = AdminSession(
            session_id=session_id,
            username=username,
            created_at=datetime.now(),
            last_activity=datetime.now(),
            ip_address=ip_address
        )

        logger.info(f"Successful login for user: {username} from IP: {ip_address}")
        return session_id

    def validate_session(self, session_id: str, ip_address: str | None = None) -> bool:
        """
        Validate if session is active and valid.

        Args:
            session_id: Session ID to validate
            ip_address: Optional IP address to verify session origin

        Returns:
            True if session is valid, False otherwise
        """
        if session_id not in self.sessions:
            return False

        session = self.sessions[session_id]

        # Check if expired
        if session.is_expired(self.session_timeout):
            logger.info(f"Session expired for user: {session.username}")
```

```
            del self.sessions[session_id]
            return False

        # Check IP if provided (optional security layer)
        if ip_address and session.ip_address != ip_address:
            logger.warning(f"IP mismatch for session: {session_id}")
            return False

        # Update activity
        session.update_activity()
        return True

    def get_session(self, session_id: str) -> AdminSession | None:
        """Get session details if valid"""
        if self.validate_session(session_id):
            return self.sessions[session_id]
        return None

    def logout(self, session_id: str) -> None:
        """Terminate session"""
        if session_id in self.sessions:
            username = self.sessions[session_id].username
            del self.sessions[session_id]
            logger.info(f"User logged out: {username}")

    def cleanup_expired_sessions(self) -> None:
        """Remove all expired sessions (should be called periodically)"""
        expired = [
            sid for sid, session in self.sessions.items()
            if session.is_expired(self.session_timeout)
        ]

        for sid in expired:
            del self.sessions[sid]

        if expired:
            logger.info(f"Cleaned up {len(expired)} expired sessions")

    def add_user(self, username: str, password: str, role: str = "user") -> None:
        """Add new user (admin function)"""
        salt = secrets.token_hex(16)
        password_hash = self._hash_password(password, salt)

        self.users[username] = {
            "password_hash": password_hash,
            "salt": salt,
            "role": role
        }

        logger.info(f"New user added: {username} with role: {role}")

     def change_password(self, username: str, old_password: str, new_password: str) ->
bool:
        """Change user password"""
```

```
        if username not in self.users:
            return False

        user = self.users[username]
        old_hash = self._hash_password(old_password, user["salt"])

        if old_hash != user["password_hash"]:
            logger.warning(f"Failed password change for user: {username}")
            return False

        # Generate new salt for additional security
        new_salt = secrets.token_hex(16)
        new_hash = self._hash_password(new_password, new_salt)

        self.users[username]["password_hash"] = new_hash
        self.users[username]["salt"] = new_salt

        logger.info(f"Password changed for user: {username}")
        return True


# Global authenticator instance
_authenticator: AdminAuthenticator | None = None


def get_authenticator() -> AdminAuthenticator:
    """Get or create global authenticator instance"""
    global _authenticator
    if _authenticator is None:
        _authenticator = AdminAuthenticator()
    return _authenticator


def require_auth(func):
    """Decorator for Flask routes requiring authentication"""
    from functools import wraps

    from flask import jsonify, request

    @wraps(func)
    def wrapper(*args, **kwargs):
        session_id = request.cookies.get('atroz_session')
        if not session_id:
            session_id = request.headers.get('X-Session-ID')

        if not session_id:
            return jsonify({"error": "Authentication required"}), 401

        auth = get_authenticator()
        ip_address = request.remote_addr

        if not auth.validate_session(session_id, ip_address):
            return jsonify({"error": "Invalid or expired session"}), 401
```

```
    return func(*args, **kwargs)

return wrapper
```