src/farfan_pipeline/dashboard_atroz_/config.py

```python
"""
Dashboard Configuration
"""
import os

# Database Configuration
# Default to a local PostgreSQL instance, or SQLite for development
DATABASE_URL = os.getenv("ATROZ_DATABASE_URL",
"postgresql://user:pass@localhost/atroz_dashboard")
USE_SQLITE = os.getenv("ATROZ_USE_SQLITE", "false").lower() == "true"

# Feature Flags
ENABLE_REALTIME_INGESTION = os.getenv("ATROZ_ENABLE_INGESTION", "true").lower() ==
"true"
```

src/farfan_pipeline/dashboard_atroz_/cross_document_api.py

```python
"""
Cross-Document Comparative Analysis API

REST API endpoints for multi-document comparative queries and analysis.
Production-ready API with comprehensive validation and error handling.
"""

from __future__ import annotations

import logging

from flask import Blueprint, jsonify, request
from pydantic import BaseModel, Field, ValidationError

from farfan_pipeline.analysis.cross_document_comparative import (
    AggregationMethod,
    ComparisonDimension,
    ComparisonOperator,
    CrossDocumentAnalyzer,
)

logger = logging.getLogger(__name__)

cross_document_bp = Blueprint(
    "cross_document", __name__, url_prefix="/api/v1/cross-document"
)


class ComparisonRequest(BaseModel):
    dimension: str = Field(..., description="Comparison dimension to use")
    aggregation_method: str = Field(default="mean", description="Aggregation method")
    policy_area_filter: list[str] | None = Field(
        default=None, description="Filter by policy areas"
    )
    dimension_filter: list[str] | None = Field(
        default=None, description="Filter by dimensions"
    )
    top_n: int | None = Field(default=None, description="Return only top N results")


class ThresholdQueryRequest(BaseModel):
    dimension: str = Field(..., description="Comparison dimension")
    operator: str = Field(
        ..., description="Comparison operator (gt, gte, lt, lte, eq, neq)"
    )
    threshold: float = Field(..., description="Threshold value")
    aggregation_method: str = Field(default="mean", description="Aggregation method")


class CausalChainQueryRequest(BaseModel):
    top_n: int = Field(default=10, description="Number of top results to return")
    min_chain_length: int = Field(default=3, description="Minimum causal chain length")
```

```python
class StatisticsRequest(BaseModel):
    dimension: str = Field(..., description="Dimension to compute statistics for")


_analyzer: CrossDocumentAnalyzer | None = None


def initialize_analyzer(analyzer: CrossDocumentAnalyzer) -> None:
    global _analyzer
    _analyzer = analyzer
    logger.info("Cross-document analyzer initialized for API")


def get_analyzer() -> CrossDocumentAnalyzer:
    if _analyzer is None:
        raise RuntimeError(
            "CrossDocumentAnalyzer not initialized. Call initialize_analyzer first."
        )
    return _analyzer


@cross_document_bp.route("/health", methods=["GET"])
def health_check():
    try:
        analyzer = get_analyzer()
        doc_count = len(analyzer.index.get_all_document_ids())
        chunk_count = len(analyzer.index.chunks)

        return (
            jsonify(
                {
                    "status": "healthy",
                    "documents_indexed": doc_count,
                    "chunks_indexed": chunk_count,
                    "policy_areas": len(analyzer.index.get_all_policy_areas()),
                    "dimensions": len(analyzer.index.get_all_dimensions()),
                }
            ),
            200,
        )
    except Exception as e:
        logger.error(f"Health check failed: {e}")
        return jsonify({"status": "unhealthy", "error": str(e)}), 503


@cross_document_bp.route("/compare", methods=["POST"])
def compare_documents():
    try:
        data = request.get_json()
        req = ComparisonRequest(**data)

        try:
            dimension = ComparisonDimension(req.dimension)
        except ValueError:
            return (
                jsonify(
                    {
                        "error": f"Invalid dimension: {req.dimension}",
                        "valid_dimensions": [d.value for d in ComparisonDimension],
                    }
                ),
                400,
            )

        try:
            aggregation = AggregationMethod(req.aggregation_method)
        except ValueError:
            return (
                jsonify(
                    {
                        "error": f"Invalid aggregation method:
{req.aggregation_method}",
                        "valid_methods": [m.value for m in AggregationMethod],
                    }
                ),
                400,
            )

        analyzer = get_analyzer()

        if req.top_n:
            result = analyzer.query_engine.find_top_documents(
                dimension=dimension,
                n=req.top_n,
                aggregation=aggregation,
                policy_area_filter=req.policy_area_filter,
                dimension_filter=req.dimension_filter,
            )
        else:
            result = analyzer.query_engine.compare_by_dimension(
                dimension=dimension,
                aggregation=aggregation,
                policy_area_filter=req.policy_area_filter,
                dimension_filter=req.dimension_filter,
            )

        return (
            jsonify(
                {
                    "query_description": result.query_description,
                    "comparison_dimension": result.comparison_dimension.value,
                    "aggregation_method": result.aggregation_method.value,
                    "total_documents": result.total_documents,
                    "total_chunks_analyzed": result.total_chunks_analyzed,
                    "timestamp": result.timestamp,
                    "metadata": result.metadata,
                    "results": [
                        {"document_id": doc_id, "score": score, "metadata": metadata}
                        for doc_id, score, metadata in result.results
                    ],
                }
            ),
            200,
        )

    except ValidationError as e:
        return jsonify({"error": "Validation error", "details": e.errors()}), 400
    except Exception as e:
        logger.error(f"Comparison failed: {e}", exc_info=True)
        return jsonify({"error": "Internal server error", "details": str(e)}), 500


@cross_document_bp.route("/highest-strategic-importance", methods=["GET"])
def highest_strategic_importance():
    try:
        n = request.args.get("n", default=10, type=int)

        analyzer = get_analyzer()
        result = analyzer.find_pdts_with_highest_strategic_importance(n=n)

        return (
            jsonify(
                {
                    "query_description": result.query_description,
                    "total_documents": result.total_documents,
                    "total_chunks_analyzed": result.total_chunks_analyzed,
                    "results": [
                        {
                            "document_id": doc_id,
                            "strategic_importance_score": score,
                            "metadata": metadata,
                        }
                        for doc_id, score, metadata in result.results
                    ],
                }
            ),
            200,
        )

    except Exception as e:
        logger.error(f"Strategic importance query failed: {e}", exc_info=True)
        return jsonify({"error": "Internal server error", "details": str(e)}), 500


@cross_document_bp.route("/strongest-causal-chains", methods=["POST"])
def strongest_causal_chains():
    try:
        data = request.get_json() or {}
        req = CausalChainQueryRequest(**data)
        analyzer = get_analyzer()
        result = analyzer.identify_municipalities_with_strongest_causal_chains(
            n=req.top_n, min_chain_length=req.min_chain_length
        )

        return (
            jsonify(
                {
                    "query_description": result.query_description,
                    "total_documents": result.total_documents,
                    "total_chunks_analyzed": result.total_chunks_analyzed,
                    "metadata": result.metadata,
                    "results": [
                        {
                            "document_id": doc_id,
                            "causal_chain_strength": score,
                            "metadata": metadata,
                        }
                        for doc_id, score, metadata in result.results
                    ],
                }
            ),
            200,
        )

    except ValidationError as e:
        return jsonify({"error": "Validation error", "details": e.errors()}), 400
    except Exception as e:
        logger.error(f"Causal chain query failed: {e}", exc_info=True)
        return jsonify({"error": "Internal server error", "details": str(e)}), 500


@cross_document_bp.route("/threshold-query", methods=["POST"])
def threshold_query():
    try:
        data = request.get_json()
        req = ThresholdQueryRequest(**data)

        try:
            dimension = ComparisonDimension(req.dimension)
        except ValueError:
            return (
                jsonify(
                    {
                        "error": f"Invalid dimension: {req.dimension}",
                        "valid_dimensions": [d.value for d in ComparisonDimension],
                    }
                ),
                400,
            )

        try:
            operator = ComparisonOperator(req.operator)
        except ValueError:
            return (
                jsonify(
                    {
                        "error": f"Invalid operator: {req.operator}",
                        "valid_operators": [o.value for o in ComparisonOperator],
                    }
                ),
                400,
            )

        try:
            aggregation = AggregationMethod(req.aggregation_method)
        except ValueError:
            return (
                jsonify(
                    {
                                                        "error":  f"Invalid  aggregation  method:
{req.aggregation_method}",
                        "valid_methods": [m.value for m in AggregationMethod],
                    }
                ),
                400,
            )

        analyzer = get_analyzer()
        result = analyzer.query_engine.find_documents_by_threshold(
            dimension=dimension,
            operator=operator,
            threshold=req.threshold,
            aggregation=aggregation,
        )

        return (
            jsonify(
                {
                    "query_description": result.query_description,
                    "total_documents": result.total_documents,
                    "total_chunks_analyzed": result.total_chunks_analyzed,
                    "metadata": result.metadata,
                    "results": [
                        {"document_id": doc_id, "score": score, "metadata": metadata}
                        for doc_id, score, metadata in result.results
                    ],
                }
            ),
            200,
        )

    except ValidationError as e:
        return jsonify({"error": "Validation error", "details": e.errors()}), 400
    except Exception as e:
        logger.error(f"Threshold query failed: {e}", exc_info=True)
        return jsonify({"error": "Internal server error", "details": str(e)}), 500
```

```
@cross_document_bp.route("/statistics", methods=["POST"])
def global_statistics():
    try:
        data = request.get_json()
        req = StatisticsRequest(**data)

        try:
            dimension = ComparisonDimension(req.dimension)
        except ValueError:
            return (
                jsonify(
                    {
                        "error": f"Invalid dimension: {req.dimension}",
                        "valid_dimensions": [d.value for d in ComparisonDimension],
                    }
                ),
                400,
            )

        analyzer = get_analyzer()
        stats = analyzer.get_global_statistics(dimension)

        return (
            jsonify(
                {
                    "dimension": stats.dimension.value,
                    "global_statistics": {
                        "mean": stats.global_mean,
                        "median": stats.global_median,
                        "std": stats.global_std,
                        "min": stats.global_min,
                        "max": stats.global_max,
                    },
                    "per_document_stats": stats.per_document_stats,
                    "per_policy_area_stats": stats.per_policy_area_stats,
                    "per_dimension_stats": stats.per_dimension_stats,
                    "total_documents": stats.total_documents,
                    "total_chunks": stats.total_chunks,
                }
            ),
            200,
        )

    except ValidationError as e:
        return jsonify({"error": "Validation error", "details": e.errors()}), 400
    except Exception as e:
        logger.error(f"Statistics query failed: {e}", exc_info=True)
        return jsonify({"error": "Internal server error", "details": str(e)}), 500


@cross_document_bp.route("/compare-policy-areas", methods=["GET"])
def compare_policy_areas():
    try:
        dimension_str = request.args.get("dimension", default="strategic_importance")
        aggregation_str = request.args.get("aggregation", default="mean")

        try:
            dimension = ComparisonDimension(dimension_str)
        except ValueError:
            return (
                jsonify(
                    {
                        "error": f"Invalid dimension: {dimension_str}",
                        "valid_dimensions": [d.value for d in ComparisonDimension],
                    }
                ),
                400,
            )

        try:
            aggregation = AggregationMethod(aggregation_str)
        except ValueError:
            return (
                jsonify(
                    {
                        "error": f"Invalid aggregation method: {aggregation_str}",
                        "valid_methods": [m.value for m in AggregationMethod],
                    }
                ),
                400,
            )

        analyzer = get_analyzer()
        results_by_area = analyzer.query_engine.compare_policy_areas(
            dimension=dimension, aggregation=aggregation
        )

        return (
            jsonify(
                {
                    "dimension": dimension.value,
                    "aggregation_method": aggregation.value,
                    "policy_areas": {
                        area: {
                            "query_description": result.query_description,
                            "total_documents": result.total_documents,
                            "total_chunks_analyzed": result.total_chunks_analyzed,
                            "results": [
                                {
                                    "document_id": doc_id,
                                    "score": score,
                                    "metadata": metadata,
                                }
                                for doc_id, score, metadata in result.results
                            ],
                        }
                        for area, result in results_by_area.items()
                    },
                }
            ),
            200,
        )

    except Exception as e:
        logger.error(f"Policy area comparison failed: {e}", exc_info=True)
        return jsonify({"error": "Internal server error", "details": str(e)}), 500


@cross_document_bp.route("/compare-dimensions", methods=["GET"])
def compare_dimensions():
    try:
        comparison_dim_str = request.args.get(
            "comparison_dimension", default="strategic_importance"
        )
        aggregation_str = request.args.get("aggregation", default="mean")

        try:
            comparison_dimension = ComparisonDimension(comparison_dim_str)
        except ValueError:
            return (
                jsonify(
                    {
                        "error": f"Invalid comparison dimension: {comparison_dim_str}",
                        "valid_dimensions": [d.value for d in ComparisonDimension],
                    }
                ),
                400,
            )

        try:
            aggregation = AggregationMethod(aggregation_str)
        except ValueError:
            return (
                jsonify(
                    {
                        "error": f"Invalid aggregation method: {aggregation_str}",
                        "valid_methods": [m.value for m in AggregationMethod],
                    }
                ),
                400,
            )

        analyzer = get_analyzer()
        results_by_dim = analyzer.query_engine.compare_dimensions(
            comparison_dimension=comparison_dimension, aggregation=aggregation
        )

        return (
            jsonify(
                {
                    "comparison_dimension": comparison_dimension.value,
                    "aggregation_method": aggregation.value,
                    "dimensions": {
                        dim: {
                            "query_description": result.query_description,
                            "total_documents": result.total_documents,
                            "total_chunks_analyzed": result.total_chunks_analyzed,
                            "results": [
                                {
                                    "document_id": doc_id,
                                    "score": score,
                                    "metadata": metadata,
                                }
                                for doc_id, score, metadata in result.results
                            ],
                        }
                        for dim, result in results_by_dim.items()
                    },
                }
            ),
            200,
        )

    except Exception as e:
        logger.error(f"Dimension comparison failed: {e}", exc_info=True)
        return jsonify({"error": "Internal server error", "details": str(e)}), 500


@cross_document_bp.route("/index-info", methods=["GET"])
def index_info():
    try:
        analyzer = get_analyzer()

        return (
            jsonify(
                {
                    "total_documents": len(analyzer.index.get_all_document_ids()),
                    "total_chunks": len(analyzer.index.chunks),
                    "policy_areas": sorted(list(analyzer.index.get_all_policy_areas())),
                    "dimensions": sorted(list(analyzer.index.get_all_dimensions())),
                    "document_ids": sorted(analyzer.index.get_all_document_ids()),
                    "index_hash": analyzer.index.compute_index_hash(),
                    "last_update": (
                        analyzer.index._last_update.isoformat()
                        if analyzer.index._last_update
                        else None
                    ),
                }
            ),
            200,
        )

    except Exception as e:
        logger.error(f"Index info retrieval failed: {e}", exc_info=True)
        return jsonify({"error": "Internal server error", "details": str(e)}), 500
```

src/farfan_pipeline/dashboard_atroz_/dashboard_data_service.py

```python
from __future__ import annotations

import json
import logging
from datetime import datetime
from pathlib import Path
from typing import Any, Iterable, Sequence

logger = logging.getLogger(__name__)


class DashboardDataService:
    """Transforms pipeline artifacts into dashboard-friendly payloads."""

    def __init__(self, jobs_dir: Path) -> None:
        self.jobs_dir = jobs_dir

    # ---------------------------------------------------------------------
    # Public interface
    # ---------------------------------------------------------------------

    def summarize_region(
        self,
        record: dict[str, Any],
    ) -> tuple[dict[str, Any], dict[str, Any]]:
        """Build lightweight region summary and context for downstream use."""
        report = self._load_report(record)
        macro_detail = self._extract_macro_detail(report, record)
        clusters = self._extract_clusters(report, record)
        scores = self._build_scores(record, macro_detail)
        indicators = self._build_indicators(record, macro_detail, scores, clusters)
        metadata = self._build_metadata(record, macro_detail)

        summary = {
            'id': record.get('id'),
            'job_id': record.get('job_id'),
            'name': (record.get('name') or '').upper(),
            'municipality': record.get('municipality'),
            'coordinates': self._build_coordinates(record),
            'metadata': metadata,
            'scores': scores,
              'clusterScores': {entry['cluster_id']: entry['score_percent'] for entry in
clusters if entry.get('cluster_id')},
            'connections': list(record.get('connections') or []),
            'indicators': indicators,
            'macroBand': metadata.get('macroBand'),
            'updated_at': record.get('updated_at'),
        }

        context = {
            'report': report,
            'macro': macro_detail,
            'clusters': clusters,
        }
        return summary, context

    def build_region_detail(
        self,
        record: dict[str, Any],
        summary: dict[str, Any],
        context: dict[str, Any],
        region_evidence: Iterable[dict[str, Any]] | None = None,
    ) -> dict[str, Any]:
        """Build full region payload with macro/meso/micro breakdown."""
        report = context.get('report') or {}
             macro_detail  =  context.get('macro')  or  self._extract_macro_detail(report,
record)
        clusters = context.get('clusters') or self._extract_clusters(report, record)
        question_matrix = self._extract_question_matrix(report)
        recommendations = self._extract_recommendations(report, macro_detail, clusters)
        evidence_stream = self._merge_evidence(region_evidence, question_matrix, record)

        detailed = dict(summary)
        detailed['macro'] = macro_detail
        detailed['meso'] = clusters
        detailed['micro'] = question_matrix
        detailed['detailed_analysis'] = {
            'cluster_breakdown': self._to_cluster_breakdown(clusters),
            'question_matrix': question_matrix,
            'recommendations': recommendations,
            'evidence': evidence_stream,
        }
        return detailed

    def extract_question_matrix(self, report: dict[str, Any]) -> list[dict[str, Any]]:
        """Expose normalized question matrix for other services."""
        return self._extract_question_matrix(report)

    def normalize_evidence_stream(
        self,
        evidence: Iterable[dict[str, Any]],
        limit: int = 50,
    ) -> list[dict[str, Any]]:
        """Normalize persisted evidence items for ticker display."""
        normalized = [self._normalize_evidence_item(item) for item in evidence if item]
        normalized = [item for item in normalized if item]
        normalized.sort(key=lambda item: item.get('timestamp') or '', reverse=True)
        return normalized[:limit]

    # ------------------------------------------------------------------
    # Report parsing helpers
    # ------------------------------------------------------------------

    def _load_report(self, record: dict[str, Any]) -> dict[str, Any]:
        """Load latest pipeline report for region if available."""
        job_id = record.get('latest_report') or record.get('job_id')
        candidate_paths: list[Path] = []

        report_path = record.get('report_path')
        if report_path:
            candidate_paths.append(Path(report_path))
        if job_id:
            candidate_paths.append(self.jobs_dir / f"{job_id}.json")

        for path in candidate_paths:
            if not path or not path.exists():
                continue
            try:
                with open(path, 'r', encoding='utf-8') as handle:
                    payload = json.load(handle)
                if isinstance(payload, dict):
                    report = payload.get('report') if 'report' in payload else payload
                    if isinstance(report, dict):
                        return report
            except Exception as exc:  # pragma: no cover - best effort
                logger.warning("Failed to load dashboard report %s: %s", path, exc)
        return {}

    def _build_coordinates(self, record: dict[str, Any]) -> dict[str, float]:
        coords = record.get('coordinates') or {}
         x = self._first_number([coords.get('x'), coords.get('lng'), coords.get('lon')],
default=50.0)
        y = self._first_number([coords.get('y'), coords.get('lat')], default=50.0)
        return {'x': float(x), 'y': float(y)}

     def _build_metadata(self, record: dict[str, Any], macro_detail: dict[str, Any]) ->
dict[str, Any]:
        stats = dict(record.get('stats') or {})
        if 'departments' in stats and not isinstance(stats['departments'], list):
            stats['departments'] = list(self._ensure_list(stats['departments']))
        if macro_detail.get('band'):
            stats['macroBand'] = macro_detail['band']
        return stats

      def _build_scores(self, record: dict[str, Any], macro_detail: dict[str, Any]) ->
dict[str, Any]:
        scores = dict(record.get('scores') or {})
        if macro_detail.get('score_percent') is not None:
            scores.setdefault('overall', macro_detail['score_percent'])
        scores.setdefault('lastUpdated', record.get('updated_at'))
        return scores

    def _build_indicators(
        self,
        record: dict[str, Any],
        macro_detail: dict[str, Any],
        scores: dict[str, Any],
        clusters: Sequence[dict[str, Any]],
    ) -> dict[str, Any]:
        if record.get('indicators'):
            return dict(record['indicators'])

            cluster_map = {entry['cluster_id'].lower(): entry for entry in clusters if
entry.get('cluster_id')}
        alignment = macro_detail.get('score_percent')
        implementation = None
        impact = None
        if 'cl01' in cluster_map:
            implementation = cluster_map['cl01'].get('score_percent')
        implementation = implementation or scores.get('governance')
        if 'cl04' in cluster_map:
            impact = cluster_map['cl04'].get('score_percent')
        impact = impact or scores.get('environmental') or scores.get('social')

        return {
            'alignment': alignment,
            'implementation': implementation,
            'impact': impact,
        }

    def _extract_macro_detail(self, report: dict[str, Any], record: dict[str, Any]) ->
dict[str, Any]:
        macro_section: dict[str, Any] = {}
        if isinstance(report, dict):
            for key in ('macro_analysis', 'macro_summary', 'macro'):
                candidate = report.get(key)
                if isinstance(candidate, dict):
                    macro_section = candidate
                    break
        raw_scores = record.get('raw_scores') or {}
        percent_scores = record.get('scores') or {}

        score = self._first_number(
            [
                macro_section.get('overall_score'),
                macro_section.get('overall_posterior'),
                macro_section.get('adjusted_score'),
                raw_scores.get('overall'),
                self._maybe_percentage_to_fraction(percent_scores.get('overall')),
            ]
        )
        score_percent = self._first_number(
            [
                percent_scores.get('overall'),
                macro_section.get('overall_score_percent'),
                self._to_percent(score),
            ]
        )
          band = macro_section.get('quality_band') or macro_section.get('quality_level')
or record.get('macro_band')
        coherence = self._first_number(
            [
                macro_section.get('cross_cutting_coherence'),
                macro_section.get('coherence'),
                macro_section.get('coherence_index'),
                                  macro_section.get('metadata',  {}).get('coherence')  if
isinstance(macro_section.get('metadata'), dict) else None,
            ]
        )
        systemic_gaps = macro_section.get('systemic_gaps')
        if systemic_gaps is None and isinstance(macro_section.get('metadata'), dict):
            systemic_gaps = macro_section['metadata'].get('systemic_gaps')
        systemic_gaps = list(self._ensure_list(systemic_gaps)) if systemic_gaps else []
        alignment = self._first_number(
            [
                macro_section.get('strategic_alignment'),
                macro_section.get('alignment'),
            ]
        )

        return {
            'score': score,
            'score_percent': score_percent,
            'band': band,
            'coherence': coherence,
            'systemic_gaps': systemic_gaps,
            'alignment': alignment,
            'updated_at': record.get('updated_at'),
        }

    def _extract_clusters(
        self,
        report: dict[str, Any],
        record: dict[str, Any],
    ) -> list[dict[str, Any]]:
        """Normalize cluster structures from report/state."""
        clusters: list[dict[str, Any]] = []
        cluster_section: Any = None

        if isinstance(report, dict):
            meso_section = report.get('meso_analysis')
            if isinstance(meso_section, dict):
                                cluster_section  =  meso_section.get('cluster_scores')  or
meso_section.get('clusters')
            if cluster_section is None:
                cluster_section = report.get('meso_clusters')

        if isinstance(cluster_section, dict):
            for key, value in cluster_section.items():
                clusters.append(self._normalize_cluster_entry(value, key))
        elif isinstance(cluster_section, list):
            for entry in cluster_section:
                clusters.append(self._normalize_cluster_entry(entry))
        elif record.get('cluster_details'):
            for entry in record['cluster_details']:
                if isinstance(entry, dict):
                    clusters.append(self._normalize_cluster_entry(entry))
        if not clusters and record.get('cluster_scores'):
            for key, value in record['cluster_scores'].items():
                score_percent = self._sanitize_percent(value)
                score = self._maybe_percentage_to_fraction(score_percent)
                clusters.append(
                    {
                        'cluster_id': key,
                        'name': str(key).upper(),
                        'score': score,
                        'score_percent': score_percent,
                        'coherence': None,
                        'variance': None,
                        'areas': [],
                        'weakest_area': None,
                        'trend': 0.0,
                    }
                )

        clusters = [entry for entry in clusters if entry.get('cluster_id')]
        clusters.sort(key=lambda entry: entry.get('cluster_id'))
        return clusters

    def _normalize_cluster_entry(
        self,
        entry: Any,
        fallback_id: str | None = None,
    ) -> dict[str, Any]:
        if not isinstance(entry, dict):
            return {}
        cluster_id = entry.get('cluster_id') or entry.get('id') or fallback_id
        if not cluster_id:
            return {}
        name = entry.get('cluster_name') or entry.get('name') or str(cluster_id)
        score = self._first_number(
            [
                entry.get('adjusted_score'),
                entry.get('score'),
                entry.get('raw_meso_score'),
                entry.get('value'),
            ]
        )
        score_percent = self._first_number(
            [
                entry.get('score_percent'),
                self._to_percent(score),
            ]
        )
        coherence = self._first_number(
            [
                entry.get('coherence'),
                                          entry.get('metadata',  {}).get('coherence')  if
isinstance(entry.get('metadata'), dict) else None,
                                 entry.get('dispersion_metrics',  {}).get('coherence')  if
isinstance(entry.get('dispersion_metrics'), dict) else None,
            ]
        )
        variance = self._first_number(
            [
                entry.get('variance'),
                entry.get('dispersion_penalty'),
                                          entry.get('metadata',  {}).get('variance')  if
isinstance(entry.get('metadata'), dict) else None,
            ]
        )
        areas = self._extract_cluster_areas(entry)
        weakest_area = entry.get('weakest_area')
        if not weakest_area and isinstance(entry.get('metadata'), dict):
            weakest_area = entry['metadata'].get('weakest_area')
        if not weakest_area and areas:
            weakest_area = areas[0]
        trend = self._first_number(
            [
                entry.get('trend'),
                                             entry.get('metadata',  {}).get('trend')  if
isinstance(entry.get('metadata'), dict) else None,
            ],
            default=0.0,
        )

        return {
            'cluster_id': str(cluster_id),
            'name': str(name).upper(),
            'score': score,
            'score_percent': score_percent,
            'coherence': coherence,
            'variance': variance,
            'areas': areas,
            'weakest_area': weakest_area,
            'trend': trend,
        }

    def _extract_cluster_areas(self, entry: dict[str, Any]) -> list[str]:
        areas_field = entry.get('areas')
        if isinstance(areas_field, list):
            return [str(area) for area in areas_field]
        area_scores = entry.get('area_scores')
        if isinstance(area_scores, list):
            names: list[str] = []
            for area in area_scores:
                if not isinstance(area, dict):
                    continue
                    candidate = area.get('policy_area_id') or area.get('policy_area') or
area.get('name')
                if candidate:
                    names.append(str(candidate))
            return names
        return []
    def _extract_question_matrix(self, report: dict[str, Any]) -> list[dict[str, Any]]:
        questions_raw: list[dict[str, Any]] = []
        micro_section = report.get('micro_analysis') if isinstance(report, dict) else
None
        if isinstance(micro_section, dict):
            if isinstance(micro_section.get('questions'), list):
                questions_raw.extend([item for item in micro_section['questions'] if
isinstance(item, dict)])
            question_scores = micro_section.get('question_scores')
            if isinstance(question_scores, dict):
                for key, value in question_scores.items():
                    questions_raw.append({'question_id': key, 'score': value})
        micro_analyses = report.get('micro_analyses') if isinstance(report, dict) else
None
        if isinstance(micro_analyses, list):
            questions_raw.extend([item for item in micro_analyses if isinstance(item,
dict)])

        normalized: list[dict[str, Any]] = []
        for item in questions_raw:
            normalized_item = self._normalize_question_entry(item)
            if normalized_item:
                normalized.append(normalized_item)
        return normalized

    def _normalize_question_entry(self, item: dict[str, Any]) -> dict[str, Any] | None:
        question_id = item.get('question_id') or item.get('id') or item.get('code')
        if not question_id:
            return None
        text = item.get('text')
        if not text and isinstance(item.get('metadata'), dict):
            text = item['metadata'].get('title') or
item['metadata'].get('question_text')
        text = text or f'Pregunta {question_id}'

        metadata = item.get('metadata') if isinstance(item.get('metadata'), dict) else
{}
        policy_area = metadata.get('policy_area_id') or metadata.get('policy_area')
        dimension = metadata.get('dimension_id') or metadata.get('dimension')
        if not policy_area or not dimension:
            inferred_area, inferred_dimension =
self._parse_question_identifier(str(question_id))
            policy_area = policy_area or inferred_area
            dimension = dimension or inferred_dimension

        score = self._first_number([
            item.get('score'),
            item.get('adjusted_score'),
            item.get('value'),
        ])
        score_percent = None
        if score is not None:
            if score <= 1.0:
                score_percent = self._to_percent(score)
            elif score <= 3.0:
                score_percent = self._sanitize_percent((score / 3.0) * 100.0)
            else:
                score_percent = self._sanitize_percent(score)

        evidence = self._normalize_evidence_list(item.get('evidence'), question_id)
         recommendations = self._normalize_recommendations(item.get('recommendations') or
item.get('recommendation'))

        return {
            'id': str(question_id),
            'text': text,
            'score': score,
            'score_percent': score_percent,
            'category': policy_area,
            'dimension': dimension,
            'evidence': evidence,
            'recommendations': recommendations,
        }

     def  _parse_question_identifier(self,  identifier:  str)  ->  tuple[str  |  None,  str  |
None]:
        cleaned = identifier.replace('_', '-').upper()
        parts = cleaned.split('-')
        policy_area = None
        dimension = None
        for part in parts:
            if part.startswith('PA') and part[2:].isdigit():
                policy_area = part
            elif part.startswith('DIM') and part[3:].isdigit():
                dimension = part
            elif part.startswith('D') and part[1:].isdigit() and not dimension:
                dimension = f'DIM{part[1:]}'
        return policy_area, dimension

    def _normalize_recommendations(self, value: Any) -> list[str]:
        if value is None:
            return []
        if isinstance(value, list):
            return [str(item) for item in value]
        return [str(value)]

    def _extract_recommendations(
        self,
        report: dict[str, Any],
        macro_detail: dict[str, Any],
        clusters: Sequence[dict[str, Any]],
    ) -> list[dict[str, Any]]:
        raw_recs = report.get('recommendations') if isinstance(report, dict) else None
        if isinstance(raw_recs, dict):
            items = raw_recs.get('items')
            raw_recs = items if isinstance(items, list) else list(raw_recs.values())
        recommendations: list[dict[str, Any]] = []
        if isinstance(raw_recs, list):
            for entry in raw_recs:
                normalized = self._normalize_recommendation_entry(entry)
                if normalized:
                    recommendations.append(normalized)
        elif raw_recs:
            normalized = self._normalize_recommendation_entry(raw_recs)
            if normalized:
                recommendations.append(normalized)

        if not recommendations:
            for gap in macro_detail.get('systemic_gaps') or []:
                recommendations.append(
                    {
                        'priority': 'ALTA',
                        'text': str(gap),
                        'category': 'MACRO',
                        'impact': 'HIGH',
                    }
                )
        if not recommendations and clusters:
            worst = min(
                 (cluster for cluster in clusters if cluster.get('score_percent') is not
None),
                key=lambda cluster: cluster['score_percent'],
                default=None,
            )
            if worst:
                recommendations.append(
                    {
                        'priority': 'MEDIA',
                        'text': f'Reforzar intervenciones en {worst["name"]}',
                        'category': 'MESO',
                        'impact': 'MEDIUM',
                    }
                )
        return recommendations

    def _normalize_recommendation_entry(self, entry: Any) -> dict[str, Any] | None:
        if isinstance(entry, dict):
            text = entry.get('description') or entry.get('text') or entry.get('message')
            if not text:
                return None
                 severity  =  str(entry.get('severity')  or  entry.get('priority')  or
'MEDIUM').upper()
            priority = self._severity_to_priority(severity)
                        category  =  str(entry.get('category')  or  entry.get('type')  or
entry.get('source') or 'GENERAL').upper()
            impact = str(entry.get('impact') or severity).upper()
            return {
                'priority': priority,
                'text': text,
                'category': category,
                'impact': impact,
            }
        if entry:
            return {
                'priority': 'MEDIA',
                'text': str(entry),
                'category': 'GENERAL',
                'impact': 'MEDIUM',
            }
        return None

    def _merge_evidence(
        self,
        region_evidence: Iterable[dict[str, Any]] | None,
        question_matrix: Sequence[dict[str, Any]],
        record: dict[str, Any],
    ) -> list[dict[str, Any]]:
        merged: list[dict[str, Any]] = []
        seen: set[tuple[Any, ...]] = set()
        region_id = record.get('id')
        timestamp = record.get('updated_at') or datetime.now().isoformat()

        if region_evidence:
            for item in region_evidence:
                                      normalized  =  self._normalize_evidence_item(item,
default_region_id=region_id)
                if not normalized:
                    continue
                key = (
                    normalized.get('source'),
                    normalized.get('page'),
                    normalized.get('text'),
                    normalized.get('question_id'),
                )
                if key in seen:
                    continue
                seen.add(key)
                merged.append(normalized)

        for question in question_matrix:
            for evidence in question.get('evidence') or []:
                normalized = dict(evidence)
                normalized.setdefault('region_id', region_id)
                normalized.setdefault('question_id', question.get('id'))
                normalized.setdefault('timestamp', timestamp)
                key = (
                    normalized.get('source'),
                    normalized.get('page'),
                    normalized.get('text'),
                    normalized.get('question_id'),
                )
                if key in seen:
                    continue
                seen.add(key)
                merged.append(normalized)
        merged.sort(key=lambda item: item.get('timestamp') or '', reverse=True)
        return merged[:200]

    def _normalize_evidence_list(
        self,
        evidence: Any,
        question_id: str | None,
    ) -> list[dict[str, Any]]:
        if evidence is None:
            return []
        items = evidence if isinstance(evidence, list) else [evidence]
        normalized: list[dict[str, Any]] = []
        for entry in items:
            normalized_entry = self._normalize_evidence_item(entry)
            if normalized_entry:
                normalized_entry.setdefault('question_id', question_id)
                normalized.append(normalized_entry)
        return normalized

    def _normalize_evidence_item(
        self,
        evidence: Any,
        default_region_id: str | None = None,
    ) -> dict[str, Any] | None:
        if isinstance(evidence, dict):
            source = evidence.get('source') or evidence.get('document') or 'Desconocido'
            page = self._safe_int(evidence.get('page') or evidence.get('page_number'))
                          text  =  evidence.get('text')  or  evidence.get('excerpt')  or
evidence.get('content')
            if not text:
                return None
            timestamp = evidence.get('timestamp') or datetime.now().isoformat()
            normalized: dict[str, Any] = {
                'source': str(source),
                'page': page,
                'text': text,
                'timestamp': timestamp,
                'region_id': evidence.get('region_id') or default_region_id,
            }
            if 'relevance' in evidence:
                                                             normalized['relevance']    =
self._first_number([evidence.get('relevance')])
            if 'job_id' in evidence:
                normalized['job_id'] = evidence['job_id']
            if 'question_id' in evidence:
                normalized['question_id'] = evidence['question_id']
            return normalized
        if isinstance(evidence, str):
            return {
                'source': 'Documento',
                'page': None,
                'text': evidence,
                'timestamp': datetime.now().isoformat(),
                'region_id': default_region_id,
            }
        return None

    def _to_cluster_breakdown(self, clusters: Sequence[dict[str, Any]]) ->
list[dict[str, Any]]:
        breakdown: list[dict[str, Any]] = []
        for cluster in clusters:
            value = cluster.get('score_percent')
            if value is None and cluster.get('score') is not None:
                value = self._to_percent(cluster['score'])
            breakdown.append(
                {
                    'name': cluster.get('name') or cluster.get('cluster_id'),
                    'value': value,
                    'trend': cluster.get('trend', 0.0),
                    'weakest_area': cluster.get('weakest_area'),
                }
            )
        return breakdown

    # ------------------------------------------------------------------
    # Utility helpers
    # ------------------------------------------------------------------

    def _severity_to_priority(self, severity: str) -> str:
        mapping = {
            'CRITICAL': 'CRITICA',
            'HIGH': 'ALTA',
            'MEDIUM': 'MEDIA',
            'LOW': 'BAJA',
        }
        return mapping.get(severity.upper(), severity.upper())

    def _safe_int(self, value: Any) -> int | None:
        try:
            if value is None:
                return None
            return int(value)
        except (TypeError, ValueError):
            return None

    def _first_number(self, candidates: Sequence[Any], default: float | None = None) ->
float | None:
        for candidate in candidates:
            try:
                if candidate is None:
                    continue
                value = float(candidate)
                if not (value != value):  # NaN check
                    return value
            except (TypeError, ValueError):
                continue
        return default
```

```
def _maybe_percentage_to_fraction(self, value: Any) -> float | None:
    if value is None:
        return None
    try:
        numeric = float(value)
    except (TypeError, ValueError):
        return None
    if numeric > 1.0:
        return numeric / 100.0
    return numeric

def _sanitize_percent(self, value: Any) -> float | None:
    if value is None:
        return None
    try:
        numeric = float(value)
    except (TypeError, ValueError):
        return None
    return round(numeric, 2)

def _to_percent(self, value: Any) -> float | None:
    numeric = self._first_number([value])
    if numeric is None:
        return None
    if numeric <= 1.0:
        return round(numeric * 100.0, 2)
    return round(numeric, 2)

def _ensure_list(self, value: Any) -> Iterable[Any]:
    if value is None:
        return []
    if isinstance(value, list):
        return value
    if isinstance(value, tuple):
        return list(value)
    return [value]
```

src/farfan_pipeline/dashboard_atroz_/dashboard_server.py

```python
import os
import time
import json
import logging
from typing import Dict, Any, List
from flask import Flask, jsonify, request, send_from_directory
from flask_cors import CORS
from flask_socketio import SocketIO, emit
from werkzeug.utils import secure_filename
from orchestration.orchestrator import Orchestrator

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger("atroz_dashboard")

# Initialize Flask App
PROJECT_ROOT = '/home/recovered/F.A.R.F.A.N-MECHANISTIC_POLICY_PIPELINE_FINAL-3'
app = Flask(__name__, static_folder=PROJECT_ROOT, static_url_path='')
app.config['SECRET_KEY'] = os.getenv('MANIFEST_SECRET_KEY', 'atroz-secret-key')
app.config['UPLOAD_FOLDER'] = os.path.abspath('data/uploads')
app.config['MAX_CONTENT_LENGTH'] = 50 * 1024 * 1024  # 50MB max upload

# Enable CORS for development
CORS(app)

# Initialize SocketIO
socketio = SocketIO(app, cors_allowed_origins="*", async_mode='gevent')

# Ensure upload directory exists
os.makedirs(app.config['UPLOAD_FOLDER'], exist_ok=True)

# Global state
pipeline_status = {
    "active_jobs": [],
    "completed_jobs": [],
    "system_metrics": {
        "cpu_usage": 0,
        "memory_usage": 0,
        "uptime": 0
    }
}

# Mock PDET Data (will be replaced by real data adapter)
PDET_REGIONS = [
    {"id": "arauca", "name": "Arauca", "score": 85, "x": 15, "y": 20, "municipalities": 7},
    {"id": "catatumbo", "name": "Catatumbo", "score": 72, "x": 25, "y": 15, "municipalities": 8},
    {"id": "montes_maria", "name": "Montes de María", "score": 91, "x": 35, "y": 10, "municipalities": 15},
    {"id": "pacifico_medio", "name": "Pacífico Medio", "score": 64, "x": 10, "y": 40, "municipalities": 4},
    {"id": "putumayo", "name": "Putumayo", "score": 78, "x": 20, "y": 80, "municipalities": 9},
    {"id": "sierra_nevada", "name": "Sierra Nevada", "score": 88, "x": 40, "y": 5, "municipalities": 8},
    {"id": "uraba", "name": "Urabá Antioqueño", "score": 69, "x": 20, "y": 25, "municipalities": 8},
    {"id": "choco", "name": "Chocó", "score": 55, "x": 8, "y": 30, "municipalities": 12},
    {"id": "macarena", "name": "Macarena - Guaviare", "score": 82, "x": 45, "y": 50, "municipalities": 12},
    {"id": "pacifico_nariñense", "name": "Pacífico Nariñense", "score": 60, "x": 5, "y": 70, "municipalities": 11},
    {"id": "cuenca_caguan", "name": "Cuenca del Caguán", "score": 75, "x": 30, "y": 60, "municipalities": 6},
    {"id": "sur_tolima", "name": "Sur del Tolima", "score": 89, "x": 25, "y": 45, "municipalities": 4},
    {"id": "sur_bolivar", "name": "Sur de Bolívar", "score": 67, "x": 30, "y": 20, "municipalities": 7},
    {"id": "bajo_cauca", "name": "Bajo Cauca", "score": 71, "x": 28, "y": 22, "municipalities": 13},
    {"id": "sur_cordoba", "name": "Sur de Córdoba", "score": 63, "x": 22, "y": 18, "municipalities": 5},
    {"id": "alto_patia", "name": "Alto Patía", "score": 80, "x": 15, "y": 65, "municipalities": 24}
]

# Evidence stream - will be populated by pipeline analysis
EVIDENCE_STREAM = [
    {"source": "PDT Sección 3.2", "page": 45, "text": "Implementación de estrategias municipales", "region": "arauca"},
    {"source": "PDT Capítulo 4", "page": 67, "text": "Articulación con Decálogo DDHH", "region": "catatumbo"},
    {"source": "Anexo Técnico", "page": 112, "text": "Indicadores de cumplimiento territorial", "region": "montes_maria"},
    {"source": "PDT Sección 5.1", "page": 89, "text": "Proyección territorial 2030", "region": "putumayo"},
    {"source": "PATR Capítulo 2", "page": 34, "text": "Cadenas de valor agropecuarias", "region": "bajo_cauca"},
    {"source": "PDT Sección 6.3", "page": 156, "text": "Mecanismos de participación ciudadana", "region": "choco"},
]

from flask import Flask, jsonify, request, Response

@app.route('/')
def index():
    dashboard_path = os.path.join(PROJECT_ROOT, 'dashboard.html')
    with open(dashboard_path, 'r', encoding='utf-8') as f:
        return Response(f.read(), mimetype='text/html')

@app.route('/api/pdet-regions', methods=['GET'])
def get_pdet_regions():
    """Return PDET regions with current scores"""
    return jsonify(PDET_REGIONS)
```

```
@app.route('/api/evidence', methods=['GET'])
def get_evidence():
    """Return current evidence stream"""
    return jsonify(EVIDENCE_STREAM)

@app.route('/api/upload/plan', methods=['POST'])
def upload_plan():
    """Handle PDF plan upload"""
    if 'file' not in request.files:
        return jsonify({"error": "No file part"}), 400

    file = request.files['file']
    if file.filename == '':
        return jsonify({"error": "No selected file"}), 400

    if file and file.filename.endswith('.pdf'):
        filename = secure_filename(file.filename)
        filepath = os.path.join(app.config['UPLOAD_FOLDER'], filename)
        file.save(filepath)

        job_id = f"job_{int(time.time())}"
        pipeline_status['active_jobs'].append({
            "id": job_id,
            "filename": filename,
            "status": "queued",
            "progress": 0,
            "phase": 0
        })

        # Emit update via WebSocket
        socketio.emit('job_created', {"job_id": job_id, "filename": filename})

        # Trigger pipeline (mock for now, will connect to real orchestrator)
        socketio.start_background_task(run_pipeline_mock, job_id, filename)

        return jsonify({"message": "File uploaded successfully", "job_id": job_id}), 202

    return jsonify({"error": "Invalid file type"}), 400

@app.route('/api/metrics', methods=['GET'])
def get_metrics():
    """Return system metrics"""
    import psutil
    metrics = {
        "cpu": psutil.cpu_percent(),
        "memory": psutil.virtual_memory().percent,
        "active_jobs": len(pipeline_status['active_jobs']),
        "uptime": time.time() - start_time
    }
    return jsonify(metrics)

# WebSocket Events
@socketio.on('connect')
```

```python
def handle_connect():
    logger.info("Client connected")
    emit('system_status', {"status": "online", "version": "1.0.0"})

def run_pipeline_mock(job_id, filename):
    """Mock pipeline execution to demonstrate UI updates"""
    logger.info(f"Starting pipeline for {job_id}")

    phases = [
        "Acquisition & Integrity",
        "Format Decomposition",
        "Text Extraction",
        "Structure Normalization",
        "Semantic Segmentation",
        "Entity Recognition",
        "Relation Extraction",
        "Policy Analysis",
        "Report Generation"
    ]

    for i, phase in enumerate(phases):
        time.sleep(2)  # Simulate work
        progress = int((i + 1) / len(phases) * 100)

        socketio.emit('pipeline_progress', {
            "job_id": job_id,
            "phase": i + 1,
            "phase_name": phase,
            "progress": progress,
            "status": "processing"
        })

    socketio.emit('pipeline_completed', {
        "job_id": job_id,
        "status": "completed",
        "result_url": f"/artifacts/{job_id}/report.json"
    })

start_time = time.time()

if __name__ == '__main__':
    logger.info("Starting AtroZ Dashboard Server...")
    socketio.run(app, host='0.0.0.0', port=5000, debug=True)
```

src/farfan_pipeline/dashboard_atroz_/data_service.py

```
"""Shim for the dashboard transformer service.

References the implementation in `farfan_pipeline.api.dashboard_data_service`.
"""

from ..api.dashboard_data_service import DashboardDataService

__all__ = ["DashboardDataService"]
```

src/farfan_pipeline/dashboard_atroz_/infrastructure/api_v2.py

```python
"""
API v2 Endpoints
High-performance endpoints serving data from the PostgreSQL aggregation pyramid.
"""
from fastapi import APIRouter, HTTPException, Depends
from typing import List, Optional, Dict
from pydantic import BaseModel
import uuid

router = APIRouter(prefix="/api/v2", tags=["dashboard"])

# Pydantic Models
class RegionSummary(BaseModel):
    id: str
    name: str
    macro_score: Optional[float]
    macro_band: Optional[str]
    coordinates: Optional[Dict[str, float]] = None

class ComparisonRequest(BaseModel):
    region_ids: List[str]

# Dependency
def get_db():
    # Yield database session
    # In production:
    # db = SessionLocal()
    # try: yield db
    # finally: db.close()
    pass

@router.get("/regions", response_model=List[RegionSummary])
async def list_regions(subregion_id: Optional[str] = None):
    """
    List all regions, optionally filtered by PDET subregion.
    Uses cached aggregation table.
    """
    # Simulate DB query result
    # In a real scenario, `db.query(Region).filter(...)`

    # Mock data for demonstration of API contract
    mock_regions = [
          RegionSummary(id="19050", name="ARGELIA", macro_score=75.5, macro_band="HIGH",
coordinates={"lat": 2.263, "lon": -77.194}),
         RegionSummary(id="19075", name="BALBOA", macro_score=62.0, macro_band="MEDIUM",
coordinates={"lat": 2.0, "lon": -77.0}),
                    RegionSummary(id="19100", name="BUENOS AIRES", macro_score=None,
macro_band=None, coordinates={"lat": 3.0, "lon": -76.0}),
    ]

    if subregion_id:
        # Basic filtering logic simulation
        return [r for r in mock_regions if int(r.id) > 19060] # Dummy logic

    return mock_regions

@router.get("/regions/{region_id}")
async def get_region_detail(region_id: str):
    """
    Get full drill-down details for a region (Macro -> Meso -> Micro).
    """
    return {
        "id": region_id,
        "detail": "Detailed analysis payload",
        "macro": {"score": 75.5, "band": "HIGH"},
        "meso": {"CL01": 80, "CL02": 70},
        "micro": []
    }

@router.post("/compare")
async def compare_regions(request: ComparisonRequest):
    """
    Compare multiple regions side-by-side.
    """
    return {
        "comparison_matrix": {rid: {"score": 70 + i} for i, rid in enumerate(request.region_ids)},
        "regions": request.region_ids
    }
```

src/farfan_pipeline/dashboard_atroz_/infrastructure/comparison_engine.py

```python
"""
Comparison Engine
Logic for calculating deltas and aggregating stats across regions.
"""
from typing import List, Dict, Any

class ComparisonEngine:
    def __init__(self, db_session=None):
        self.db = db_session

    def compare_regions(self, region_ids: List[str]) -> Dict[str, Any]:
        """
        Compare a list of regions.
        Returns a dictionary with 'matrix' and 'deltas'.
        """
        # In a real implementation, query the DB for these regions' scores
        # scores = self.db.query(...)

        # Mock scores for functionality
        scores = {rid: 50.0 + (i * 5.5) for i, rid in enumerate(region_ids)}

        matrix = {rid: {"overall_score": s, "rank": i+1} for rid, s in scores.items()}

        deltas = {}
        if len(region_ids) >= 2:
            r1, r2 = region_ids[0], region_ids[1]
            diff = scores[r1] - scores[r2]
            deltas[f"{r1}_vs_{r2}"] = {
                "diff": round(diff, 2),
                "leader": r1 if diff > 0 else r2
            }

        return {
            "matrix": matrix,
            "deltas": deltas,
            "metadata": {"count": len(region_ids)}
        }

    def compute_pdet_average(self, subregion_id: str) -> Dict[str, float]:
        """
        Compute average scores for a PDET subregion.
        Should use cached aggregates if available.
        """
        # Placeholder for aggregation query
        return {"overall_average": 65.4, "sample_size": 12}
```

src/farfan_pipeline/dashboard_atroz_/ingestion.py

```
"""Dashboard ingestion client (Phase 10 integration point).

Consumes the orchestrator context and posts a strictly identified municipal update to
the
AtroZ dashboard backend ingest endpoint.
"""

from __future__ import annotations

import asyncio
import re
from dataclasses import asdict, is_dataclass
from datetime import datetime, timezone
from difflib import get_close_matches
from pathlib import Path
from typing import Any, Mapping
from uuid import uuid4

import httpx
import structlog

from .api_v1_schemas import DashboardIngestRequest, MunicipalitySelector
from .api_v1_utils import slugify
from .pdet_colombia_data import PDET_MUNICIPALITIES, PDETMunicipality

logger = structlog.get_logger(__name__)

_DANE_RE = re.compile(r"(?<!\d)(\d{5})(?!\d)")


def _jsonable(obj: Any) -> Any:
    if is_dataclass(obj):
        return _jsonable(asdict(obj))
    if isinstance(obj, Mapping):
        return {str(k): _jsonable(v) for k, v in obj.items()}
    if isinstance(obj, (list, tuple)):
        return [_jsonable(v) for v in obj]
    if isinstance(obj, Path):
        return str(obj)
    if isinstance(obj, datetime):
        return obj.astimezone(timezone.utc).isoformat()
    if isinstance(obj, (str, int, float, bool)) or obj is None:
        return obj
    return str(obj)


def _extract_dane_code(*values: str) -> str | None:
    for value in values:
        match = _DANE_RE.search(value)
        if match:
            return match.group(1)
    return None
```

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


def _resolve_municipality_from_context(context: Mapping[str, Any]) -> PDETMunicipality |
None:
    document = context.get("document")
    input_data = getattr(document, "input_data", None) if document is not None else None

    doc_id = str(getattr(input_data, "document_id", "") or "")
    pdf_path = str(getattr(input_data, "pdf_path", "") or "")

    dane_code = _extract_dane_code(doc_id, pdf_path)
    if dane_code:
        matches = [m for m in PDET_MUNICIPALITIES if m.dane_code == dane_code]
        if len(matches) == 1:
            return matches[0]
        if len(matches) > 1:
             raise ValueError(f"Ambiguous DANE code match for {dane_code}: {len(matches)}
candidates")

    candidate_text = Path(pdf_path).stem if pdf_path else doc_id
    cleaned = slugify(candidate_text.replace("_", " ").replace("-", " "))

    dept_slugs = {slugify(m.department): m.department for m in PDET_MUNICIPALITIES}
    dept_hits = [dept for slug, dept in dept_slugs.items() if slug and slug in cleaned]
    dept = dept_hits[0] if len(dept_hits) == 1 else None

    name_candidates = []
    for m in PDET_MUNICIPALITIES:
        name_slug = slugify(m.name)
        if name_slug and name_slug in cleaned:
            if dept is None or m.department == dept:
                name_candidates.append(m)

    if len(name_candidates) == 1:
        return name_candidates[0]
    if len(name_candidates) > 1:
            raise ValueError(f"Ambiguous municipality match for '{candidate_text}':
{len(name_candidates)} candidates")

    names = [slugify(m.name) for m in PDET_MUNICIPALITIES]
    fuzzy = get_close_matches(cleaned, names, n=1, cutoff=0.85)
    if not fuzzy:
        return None

    fuzzy_name = fuzzy[0]
    fuzzy_matches = [m for m in PDET_MUNICIPALITIES if slugify(m.name) == fuzzy_name]
    if dept is not None:
        fuzzy_matches = [m for m in fuzzy_matches if m.department == dept]

    if len(fuzzy_matches) == 1:
        return fuzzy_matches[0]

    return None


class DashboardIngester:
    def __init__(
        self,
        ingest_url: str | None = None,
        auth_token: str | None = None,
        client_name: str = "dashboard-v1",
        client_version: str = "1.0.0",
        timeout_s: float = 5.0,
        max_retries: int = 3,
    ) -> None:
        import os

        self._ingest_url = ingest_url or os.getenv(
            "ATROZ_DASHBOARD_INGEST_URL", "http://localhost:8000/api/v1/data/ingest"
        )
        self._auth_token = auth_token or os.getenv("ATROZ_DASHBOARD_JWT", "")
        self._client_name = client_name
        self._client_version = client_version
        self._timeout_s = timeout_s
        self._max_retries = max_retries

    async def ingest_results(self, context: Mapping[str, Any]) -> bool:
        municipality = _resolve_municipality_from_context(context)
        if municipality is None:
            logger.error("dashboard_ingest_municipality_unresolved")
            return False

        document = context.get("document")
        input_data = getattr(document, "input_data", None) if document is not None else
None
        run_id = str(getattr(input_data, "run_id", "") or "") or
f"run_unknown_{int(datetime.now().timestamp())}"

        selector = MunicipalitySelector(
            id=_municipality_id(municipality),
            dane_code=municipality.dane_code or None,
            name=municipality.name,
            department=municipality.department,
            document_id=str(getattr(input_data, "document_id", "") or "") or None,
            pdf_path=str(getattr(input_data, "pdf_path", "") or "") or None,
        )

        macro_result = context.get("macro_result")
        cluster_scores = context.get("cluster_scores")
        policy_area_scores = context.get("policy_area_scores")
        dimension_scores = context.get("dimension_scores")
        scored_results = context.get("scored_results")
        micro_results = context.get("micro_results")
        minimal_micro_results: list[dict[str, Any]] | None = None
        if isinstance(micro_results, list):
            minimal_micro_results = [
                {
                    "question_id": getattr(r, "question_id", None),
                    "question_global": getattr(r, "question_global", None),
                    "base_slot": getattr(r, "base_slot", None),
                    "error": getattr(r, "error", None),
                    "duration_ms": getattr(r, "duration_ms", None),
                    "aborted": getattr(r, "aborted", None),
                }
                for r in micro_results
            ]

        payload = DashboardIngestRequest(
            run_id=run_id,
            timestamp=datetime.now(timezone.utc),
            municipality=selector,
            macro_result=_jsonable(macro_result) if macro_result is not None else None,
            cluster_scores=_jsonable(cluster_scores) if cluster_scores is not None else
None,
            policy_area_scores=_jsonable(policy_area_scores) if policy_area_scores is
not None else None,
            dimension_scores=_jsonable(dimension_scores) if dimension_scores is not None
else None,
            scored_results=_jsonable(scored_results) if scored_results is not None else
None,
            micro_results=minimal_micro_results,
        )

        headers = {
            "X-Atroz-Client": self._client_name,
            "X-Atroz-Version": self._client_version,
            "X-Request-ID": str(uuid4()),
            "Content-Type": "application/json",
        }
        if self._auth_token:
            headers["Authorization"] = f"Bearer {self._auth_token}"

        body = payload.model_dump(mode="json")

        for attempt in range(1, self._max_retries + 1):
            try:
                async with httpx.AsyncClient(timeout=self._timeout_s) as client:
                    response = await client.post(self._ingest_url, json=body,
headers=headers)

                if 200 <= response.status_code < 300:
                    logger.info(
                        "dashboard_ingest_ok",
                        municipality_id=selector.id,
                        run_id=run_id,
                        status_code=response.status_code,
                    )
                    return True

                           if response.status_code in {429, 500, 503} and attempt <
self._max_retries:
                    await asyncio.sleep(2**attempt)
                    continue

                logger.error(
                    "dashboard_ingest_failed",
                    municipality_id=selector.id,
                    run_id=run_id,
                    status_code=response.status_code,
                    response_text=response.text[:512],
                )
                return False

            except (httpx.TimeoutException, httpx.NetworkError) as exc:
                if attempt >= self._max_retries:
                    logger.error(
                        "dashboard_ingest_network_error",
                        municipality_id=selector.id,
                        run_id=run_id,
                        error=str(exc),
                    )
                    return False
                await asyncio.sleep(2**attempt)

        return False
```

src/farfan_pipeline/dashboard_atroz_/pdet_colombia_data.py

```
"""
PDET Colombia Complete Dataset
170 municipalities across 16 subregions
Data compiled from official government sources (2024)
"""

from dataclasses import dataclass
from enum import Enum
from typing import Any


class PDETSubregion(Enum):
    """16 PDET Subregions"""
    ALTO_PATIA = "Alto Patía y Norte del Cauca"
    ARAUCA = "Arauca"
    BAJO_CAUCA = "Bajo Cauca y Nordeste Antioqueño"
    CAGUAN = "Cuenca del Caguán y Piedemonte Caqueteño"
    CATATUMBO = "Catatumbo"
    CHOCO = "Chocó"
    MACARENA = "Macarena-Guaviare"
    MONTES_MARIA = "Montes de María"
    PACIFICO_MEDIO = "Pacífico Medio"
    PACIFICO_NARINENSE = "Pacífico y Frontera Nariñense"
    PUTUMAYO = "Putumayo"
    SIERRA_NEVADA = "Sierra Nevada - Perijá - Zona Bananera"
    SUR_BOLIVAR = "Sur de Bolívar"
    SUR_CORDOBA = "Sur de Córdoba"
    SUR_TOLIMA = "Sur del Tolima"
    URABA = "Urabá Antioqueño"


@dataclass
class PDETMunicipality:
    """Represents a PDET municipality"""
    name: str
    department: str
    subregion: PDETSubregion
    population: int = 0
    area_km2: float = 0.0
    dane_code: str = ""
    latitude: float = 0.0
    longitude: float = 0.0


# Complete PDET Municipality Dataset (170 municipalities)
PDET_MUNICIPALITIES: list[PDETMunicipality] = [
    # ALTO PATÍA Y NORTE DEL CAUCA (24 municipalities)
        PDETMunicipality("Argelia", "Cauca", PDETSubregion.ALTO_PATIA, 31000, 661.0,
"19050", 2.263, -77.194),
        PDETMunicipality("Balboa", "Cauca", PDETSubregion.ALTO_PATIA, 22000, 388.0,
"19075"),
      PDETMunicipality("Buenos Aires", "Cauca", PDETSubregion.ALTO_PATIA, 32000, 519.0,
"19100"),
    PDETMunicipality("Cajibío", "Cauca", PDETSubregion.ALTO_PATIA, 38000, 440.0,
"19110"),
    PDETMunicipality("Caldono", "Cauca", PDETSubregion.ALTO_PATIA, 31000, 249.0,
"19137"),
    PDETMunicipality("Caloto", "Cauca", PDETSubregion.ALTO_PATIA, 40000, 350.0,
"19142"),
    PDETMunicipality("Corinto", "Cauca", PDETSubregion.ALTO_PATIA, 33000, 273.0,
"19212"),
    PDETMunicipality("El Tambo", "Cauca", PDETSubregion.ALTO_PATIA, 50000, 3213.0,
"19256"),
    PDETMunicipality("Jambaló", "Cauca", PDETSubregion.ALTO_PATIA, 17000, 51.0,
"19364"),
    PDETMunicipality("Mercaderes", "Cauca", PDETSubregion.ALTO_PATIA, 21000, 604.0,
"19418"),
    PDETMunicipality("Miranda", "Cauca", PDETSubregion.ALTO_PATIA, 42000, 597.0,
"19455"),
    PDETMunicipality("Morales", "Cauca", PDETSubregion.ALTO_PATIA, 28000, 580.0,
"19473"),
    PDETMunicipality("Patía", "Cauca", PDETSubregion.ALTO_PATIA, 37000, 834.0, "19513"),
    PDETMunicipality("Piendamó", "Cauca", PDETSubregion.ALTO_PATIA, 44000, 116.0,
"19532"),
    PDETMunicipality("Santander de Quilichao", "Cauca", PDETSubregion.ALTO_PATIA, 95000,
543.0, "19693"),
    PDETMunicipality("Suárez", "Cauca", PDETSubregion.ALTO_PATIA, 20000, 364.0,
"19698"),
    PDETMunicipality("Toribío", "Cauca", PDETSubregion.ALTO_PATIA, 31000, 186.0,
"19821"),
    PDETMunicipality("Cumbitara", "Nariño", PDETSubregion.ALTO_PATIA, 16000, 600.0,
"52227"),
    PDETMunicipality("El Rosario", "Nariño", PDETSubregion.ALTO_PATIA, 12000, 558.0,
"52258"),
    PDETMunicipality("Leiva", "Nariño", PDETSubregion.ALTO_PATIA, 13000, 395.0,
"52381"),
    PDETMunicipality("Los Andes", "Nariño", PDETSubregion.ALTO_PATIA, 15000, 434.0,
"52427"),
    PDETMunicipality("Policarpa", "Nariño", PDETSubregion.ALTO_PATIA, 17000, 624.0,
"52585"),
    PDETMunicipality("Florida", "Valle del Cauca", PDETSubregion.ALTO_PATIA, 58000,
517.0, "76275"),
    PDETMunicipality("Pradera", "Valle del Cauca", PDETSubregion.ALTO_PATIA, 61000,
273.0, "76563"),

    # ARAUCA (4 municipalities)
    PDETMunicipality("Arauquita", "Arauca", PDETSubregion.ARAUCA, 45000, 3828.0,
"81065"),
    PDETMunicipality("Fortul", "Arauca", PDETSubregion.ARAUCA, 27000, 1997.0, "81300"),
    PDETMunicipality("Saravena", "Arauca", PDETSubregion.ARAUCA, 53000, 1879.0,
"81736"),
    PDETMunicipality("Tame", "Arauca", PDETSubregion.ARAUCA, 53000, 5278.0, "81794"),

    # BAJO CAUCA Y NORDESTE ANTIOQUEÑO (13 municipalities)
    PDETMunicipality("Cáceres", "Antioquia", PDETSubregion.BAJO_CAUCA, 39000, 2273.0,
"05120"),
    PDETMunicipality("Caucasia", "Antioquia", PDETSubregion.BAJO_CAUCA, 104000, 1842.0,
"05154"),
    PDETMunicipality("El Bagre", "Antioquia", PDETSubregion.BAJO_CAUCA, 53000, 1824.0,
"05250"),
    PDETMunicipality("Nechí", "Antioquia", PDETSubregion.BAJO_CAUCA, 29000, 2803.0,
"05495"),
    PDETMunicipality("Tarazá", "Antioquia", PDETSubregion.BAJO_CAUCA, 45000, 1923.0,
"05790"),
    PDETMunicipality("Zaragoza", "Antioquia", PDETSubregion.BAJO_CAUCA, 30000, 900.0,
"05895"),
    PDETMunicipality("Amalfi", "Antioquia", PDETSubregion.BAJO_CAUCA, 23000, 1224.0,
"05030"),
    PDETMunicipality("Anorí", "Antioquia", PDETSubregion.BAJO_CAUCA, 18000, 1445.0,
"05040"),
    PDETMunicipality("Remedios", "Antioquia", PDETSubregion.BAJO_CAUCA, 29000, 1985.0,
"05604"),
    PDETMunicipality("Segovia", "Antioquia", PDETSubregion.BAJO_CAUCA, 40000, 1234.0,
"05756"),
    PDETMunicipality("Valdivia", "Antioquia", PDETSubregion.BAJO_CAUCA, 20000, 1088.0,
"05854"),
    PDETMunicipality("Vegachí", "Antioquia", PDETSubregion.BAJO_CAUCA, 9000, 582.0,
"05858"),

    # CUENCA DEL CAGUÁN Y PIEDEMONTE CAQUETEÑO (17 municipalities)
    PDETMunicipality("Albania", "Caquetá", PDETSubregion.CAGUAN, 5000, 1149.0, "18029"),
    PDETMunicipality("Belén de los Andaquíes", "Caquetá", PDETSubregion.CAGUAN, 11000,
1168.0, "18094"),
    PDETMunicipality("Cartagena del Chairá", "Caquetá", PDETSubregion.CAGUAN, 35000,
12704.0, "18150"),
    PDETMunicipality("Curillo", "Caquetá", PDETSubregion.CAGUAN, 11000, 1463.0,
"18205"),
    PDETMunicipality("El Doncello", "Caquetá", PDETSubregion.CAGUAN, 25000, 1195.0,
"18247"),
    PDETMunicipality("El Paujil", "Caquetá", PDETSubregion.CAGUAN, 21000, 907.0,
"18256"),
    PDETMunicipality("Florencia", "Caquetá", PDETSubregion.CAGUAN, 180000, 2292.0,
"18001"),
    PDETMunicipality("La Montañita", "Caquetá", PDETSubregion.CAGUAN, 24000, 1462.0,
"18410"),
    PDETMunicipality("Milán", "Caquetá", PDETSubregion.CAGUAN, 11000, 940.0, "18460"),
    PDETMunicipality("Morelia", "Caquetá", PDETSubregion.CAGUAN, 4000, 1386.0, "18479"),
    PDETMunicipality("Puerto Rico", "Caquetá", PDETSubregion.CAGUAN, 36000, 15224.0,
"18592"),
    PDETMunicipality("San José del Fragua", "Caquetá", PDETSubregion.CAGUAN, 14000,
3938.0, "18610"),
    PDETMunicipality("San Vicente del Caguán", "Caquetá", PDETSubregion.CAGUAN, 64000,
24466.0, "18753"),
    PDETMunicipality("Solano", "Caquetá", PDETSubregion.CAGUAN, 22000, 42625.0,
"18756"),
    PDETMunicipality("Solita", "Caquetá", PDETSubregion.CAGUAN, 14000, 9057.0, "18785"),
    PDETMunicipality("Valparaíso", "Caquetá", PDETSubregion.CAGUAN, 16000, 1231.0,
"18860"),
    PDETMunicipality("Algeciras", "Huila", PDETSubregion.CAGUAN, 23000, 626.0, "41026"),
    # CATATUMBO (8 municipalities)
    PDETMunicipality("Convención", "Norte de Santander", PDETSubregion.CATATUMBO, 19000, 1171.0, "54206"),
    PDETMunicipality("El Carmen", "Norte de Santander", PDETSubregion.CATATUMBO, 15000, 1186.0, "54245"),
    PDETMunicipality("El Tarra", "Norte de Santander", PDETSubregion.CATATUMBO, 13000, 690.0, "54250"),
    PDETMunicipality("Hacarí", "Norte de Santander", PDETSubregion.CATATUMBO, 14000, 549.0, "54344"),
    PDETMunicipality("San Calixto", "Norte de Santander", PDETSubregion.CATATUMBO, 12000, 1155.0, "54660"),
    PDETMunicipality("Sardinata", "Norte de Santander", PDETSubregion.CATATUMBO, 26000, 1398.0, "54720"),
    PDETMunicipality("Teorama", "Norte de Santander", PDETSubregion.CATATUMBO, 19000, 1126.0, "54800"),
    PDETMunicipality("Tibú", "Norte de Santander", PDETSubregion.CATATUMBO, 48000, 2696.0, "54810"),

    # CHOCÓ (14 municipalities)
    PDETMunicipality("Acandí", "Chocó", PDETSubregion.CHOCO, 11000, 993.0, "27006"),
    PDETMunicipality("Bojayá", "Chocó", PDETSubregion.CHOCO, 11000, 1430.0, "27073"),
    PDETMunicipality("Carmen del Darién", "Chocó", PDETSubregion.CHOCO, 9000, 1995.0, "27135"),
    PDETMunicipality("Condoto", "Chocó", PDETSubregion.CHOCO, 20000, 1183.0, "27205"),
    PDETMunicipality("Istmina", "Chocó", PDETSubregion.CHOCO, 23000, 2394.0, "27361"),
    PDETMunicipality("Litoral de San Juan", "Chocó", PDETSubregion.CHOCO, 14000, 1024.0, "27413"),
    PDETMunicipality("Medio Atrato", "Chocó", PDETSubregion.CHOCO, 17000, 6815.0, "27425"),
    PDETMunicipality("Medio San Juan", "Chocó", PDETSubregion.CHOCO, 18000, 1331.0, "27430"),
    PDETMunicipality("Nóvita", "Chocó", PDETSubregion.CHOCO, 11000, 1619.0, "27491"),
    PDETMunicipality("Riosucio", "Chocó", PDETSubregion.CHOCO, 29000, 711.0, "27615"),
    PDETMunicipality("Sipí", "Chocó", PDETSubregion.CHOCO, 11000, 725.0, "27745"),
    PDETMunicipality("Unguía", "Chocó", PDETSubregion.CHOCO, 21000, 954.0, "27800"),
    PDETMunicipality("Murindó", "Antioquia", PDETSubregion.CHOCO, 4000, 1848.0, "05483"),
    PDETMunicipality("Vigía del Fuerte", "Antioquia", PDETSubregion.CHOCO, 6000, 956.0, "05873"),

    # MACARENA-GUAVIARE (12 municipalities)
    PDETMunicipality("Mapiripán", "Meta", PDETSubregion.MACARENA, 15000, 11341.0, "50325"),
    PDETMunicipality("Mesetas", "Meta", PDETSubregion.MACARENA, 9000, 1430.0, "50330"),
    PDETMunicipality("La Macarena", "Meta", PDETSubregion.MACARENA, 30000, 11229.0, "50350"),
    PDETMunicipality("Uribe", "Meta", PDETSubregion.MACARENA, 15000, 9506.0, "50686"),
    PDETMunicipality("Puerto Concordia", "Meta", PDETSubregion.MACARENA, 19000, 2077.0, "50568"),
    PDETMunicipality("Puerto Lleras", "Meta", PDETSubregion.MACARENA, 12000, 3987.0, "50577"),
    PDETMunicipality("Puerto Rico", "Meta", PDETSubregion.MACARENA, 19000, 2288.0, "50590"),
    PDETMunicipality("Vista Hermosa", "Meta", PDETSubregion.MACARENA, 22000, 7417.0,
"50711"),
    PDETMunicipality("San José del Guaviare", "Guaviare", PDETSubregion.MACARENA, 64000, 16592.0, "95001"),
    PDETMunicipality("Calamar", "Guaviare", PDETSubregion.MACARENA, 23000, 36157.0, "95015"),
    PDETMunicipality("El Retorno", "Guaviare", PDETSubregion.MACARENA, 19000, 18858.0, "95025"),
    PDETMunicipality("Miraflores", "Guaviare", PDETSubregion.MACARENA, 8000, 27183.0, "95200"),

    # MONTES DE MARÍA (15 municipalities)
    PDETMunicipality("Córdoba", "Bolívar", PDETSubregion.MONTES_MARIA, 14000, 336.0, "13212"),
    PDETMunicipality("El Carmen de Bolívar", "Bolívar", PDETSubregion.MONTES_MARIA, 76000, 954.0, "13244"),
    PDETMunicipality("El Guamo", "Bolívar", PDETSubregion.MONTES_MARIA, 10000, 179.0, "13268"),
    PDETMunicipality("María la Baja", "Bolívar", PDETSubregion.MONTES_MARIA, 52000, 550.0, "13442"),
    PDETMunicipality("San Jacinto", "Bolívar", PDETSubregion.MONTES_MARIA, 22000, 464.0, "13654"),
    PDETMunicipality("San Juan Nepomuceno", "Bolívar", PDETSubregion.MONTES_MARIA, 39000, 547.0, "13657"),
    PDETMunicipality("Zambrano", "Bolívar", PDETSubregion.MONTES_MARIA, 9000, 250.0, "13894"),
    PDETMunicipality("Chalán", "Sucre", PDETSubregion.MONTES_MARIA, 4000, 169.0, "70204"),
    PDETMunicipality("Coloso", "Sucre", PDETSubregion.MONTES_MARIA, 6000, 237.0, "70215"),
    PDETMunicipality("Los Palmitos", "Sucre", PDETSubregion.MONTES_MARIA, 21000, 321.0, "70429"),
    PDETMunicipality("Morroa", "Sucre", PDETSubregion.MONTES_MARIA, 16000, 258.0, "70473"),
    PDETMunicipality("Ovejas", "Sucre", PDETSubregion.MONTES_MARIA, 24000, 701.0, "70508"),
    PDETMunicipality("Palmito", "Sucre", PDETSubregion.MONTES_MARIA, 12000, 126.0, "70523"),
    PDETMunicipality("San Onofre", "Sucre", PDETSubregion.MONTES_MARIA, 50000, 1142.0, "70713"),
    PDETMunicipality("Tolú Viejo", "Sucre", PDETSubregion.MONTES_MARIA, 25000, 231.0, "70823"),

    # PACÍFICO MEDIO (4 municipalities)
    PDETMunicipality("Alto Baudó", "Chocó", PDETSubregion.PACIFICO_MEDIO, 35000, 1871.0, "27025"),
    PDETMunicipality("Bajo Baudó", "Chocó", PDETSubregion.PACIFICO_MEDIO, 16000, 1862.0, "27050"),
    PDETMunicipality("Medio Baudó", "Chocó", PDETSubregion.PACIFICO_MEDIO, 17000, 1803.0, "27420"),
    PDETMunicipality("Buenaventura", "Valle del Cauca", PDETSubregion.PACIFICO_MEDIO, 424000, 6297.0, "76109"),

    # PACÍFICO Y FRONTERA NARIÑENSE (11 municipalities)
    PDETMunicipality("Barbacoas", "Nariño", PDETSubregion.PACIFICO_NARINENSE, 24000,
1674.0, "52083"),
    PDETMunicipality("El Charco", "Nariño", PDETSubregion.PACIFICO_NARINENSE, 32000,
2485.0, "52250"),
    PDETMunicipality("Francisco Pizarro", "Nariño", PDETSubregion.PACIFICO_NARINENSE,
13000, 1585.0, "52317"),
    PDETMunicipality("La Tola", "Nariño", PDETSubregion.PACIFICO_NARINENSE, 7000, 421.0,
"52378"),
    PDETMunicipality("Magüí Payán", "Nariño", PDETSubregion.PACIFICO_NARINENSE, 23000,
1621.0, "52435"),
    PDETMunicipality("Mosquera", "Nariño", PDETSubregion.PACIFICO_NARINENSE, 12000,
1026.0, "52473"),
    PDETMunicipality("Olaya Herrera", "Nariño", PDETSubregion.PACIFICO_NARINENSE, 32000,
1932.0, "52490"),
    PDETMunicipality("Roberto Payán", "Nariño", PDETSubregion.PACIFICO_NARINENSE, 18000,
1333.0, "52621"),
    PDETMunicipality("Santa Bárbara", "Nariño", PDETSubregion.PACIFICO_NARINENSE, 9000,
1398.0, "52683"),
    PDETMunicipality("Tumaco", "Nariño", PDETSubregion.PACIFICO_NARINENSE, 215000,
3760.0, "52835"),
    PDETMunicipality("Ricaurte", "Nariño", PDETSubregion.PACIFICO_NARINENSE, 16000,
505.0, "52612"),

    # PUTUMAYO (9 municipalities)
    PDETMunicipality("Leguízamo", "Putumayo", PDETSubregion.PUTUMAYO, 21000, 12421.0,
"86573"),
    PDETMunicipality("Mocoa", "Putumayo", PDETSubregion.PUTUMAYO, 46000, 1260.0,
"86001"),
    PDETMunicipality("Orito", "Putumayo", PDETSubregion.PUTUMAYO, 21000, 587.0,
"86320"),
    PDETMunicipality("Puerto Asís", "Putumayo", PDETSubregion.PUTUMAYO, 63000, 2961.0,
"86568"),
    PDETMunicipality("Puerto Caicedo", "Putumayo", PDETSubregion.PUTUMAYO, 16000,
1297.0, "86569"),
    PDETMunicipality("Puerto Guzmán", "Putumayo", PDETSubregion.PUTUMAYO, 17000, 3221.0,
"86571"),
    PDETMunicipality("San Miguel", "Putumayo", PDETSubregion.PUTUMAYO, 24000, 4086.0,
"86755"),
    PDETMunicipality("Valle del Guamuéz", "Putumayo", PDETSubregion.PUTUMAYO, 49000,
1257.0, "86865"),
    PDETMunicipality("Villagarzón", "Putumayo", PDETSubregion.PUTUMAYO, 18000, 1470.0,
"86885"),

    # SIERRA NEVADA - PERIJÁ - ZONA BANANERA (15 municipalities)
    PDETMunicipality("Agustín Codazzi", "Cesar", PDETSubregion.SIERRA_NEVADA, 62000,
2048.0, "20013"),
    PDETMunicipality("Becerril", "Cesar", PDETSubregion.SIERRA_NEVADA, 18000, 690.0,
"20045"),
    PDETMunicipality("La Jagua de Ibirico", "Cesar", PDETSubregion.SIERRA_NEVADA, 22000,
720.0, "20383"),
    PDETMunicipality("La Paz", "Cesar", PDETSubregion.SIERRA_NEVADA, 26000, 1238.0,
"20400"),
    PDETMunicipality("Manaure Balcón del Cesar", "Cesar", PDETSubregion.SIERRA_NEVADA,
15000, 1047.0, "20443"),
    PDETMunicipality("Pueblo Bello", "Cesar", PDETSubregion.SIERRA_NEVADA, 14000, 612.0,
"20570"),
    PDETMunicipality("San Diego", "Cesar", PDETSubregion.SIERRA_NEVADA, 14000, 474.0,
"20621"),
    PDETMunicipality("Valledupar", "Cesar", PDETSubregion.SIERRA_NEVADA, 490000, 4493.0,
"20001"),
    PDETMunicipality("Dibulla", "La Guajira", PDETSubregion.SIERRA_NEVADA, 33000,
1774.0, "44090"),
    PDETMunicipality("Fonseca", "La Guajira", PDETSubregion.SIERRA_NEVADA, 36000, 494.0,
"44279"),
    PDETMunicipality("San Juan del Cesar", "La Guajira", PDETSubregion.SIERRA_NEVADA,
40000, 1671.0, "44650"),
    PDETMunicipality("Aracataca", "Magdalena", PDETSubregion.SIERRA_NEVADA, 42000,
1254.0, "47053"),
    PDETMunicipality("Ciénaga", "Magdalena", PDETSubregion.SIERRA_NEVADA, 104000,
1237.0, "47189"),
    PDETMunicipality("Fundación", "Magdalena", PDETSubregion.SIERRA_NEVADA, 63000,
988.0, "47288"),
    PDETMunicipality("Santa Marta", "Magdalena", PDETSubregion.SIERRA_NEVADA, 500000,
2381.0, "47001"),

    # SUR DE BOLÍVAR (7 municipalities)
    PDETMunicipality("Arenal", "Bolívar", PDETSubregion.SUR_BOLIVAR, 18000, 627.0,
"13052"),
    PDETMunicipality("Cantagallo", "Bolívar", PDETSubregion.SUR_BOLIVAR, 12000, 1202.0,
"13140"),
    PDETMunicipality("Morales", "Bolívar", PDETSubregion.SUR_BOLIVAR, 17000, 416.0,
"13468"),
    PDETMunicipality("San Pablo", "Bolívar", PDETSubregion.SUR_BOLIVAR, 44000, 979.0,
"13667"),
    PDETMunicipality("Santa Rosa del Sur", "Bolívar", PDETSubregion.SUR_BOLIVAR, 39000,
1749.0, "13688"),
    PDETMunicipality("Simití", "Bolívar", PDETSubregion.SUR_BOLIVAR, 20000, 2814.0,
"13744"),

    # SUR DE CÓRDOBA (5 municipalities)
    PDETMunicipality("Montelíbano", "Córdoba", PDETSubregion.SUR_CORDOBA, 83000, 2515.0,
"23466"),
    PDETMunicipality("Puerto Libertador", "Córdoba", PDETSubregion.SUR_CORDOBA, 39000,
2903.0, "23570"),
    PDETMunicipality("San José de Uré", "Córdoba", PDETSubregion.SUR_CORDOBA, 12000,
1298.0, "23682"),
    PDETMunicipality("Tierralta", "Córdoba", PDETSubregion.SUR_CORDOBA, 101000, 5084.0,
"23807"),
    PDETMunicipality("Valencia", "Córdoba", PDETSubregion.SUR_CORDOBA, 41000, 752.0,
"23855"),

    # SUR DEL TOLIMA (4 municipalities)
    PDETMunicipality("Ataco", "Tolima", PDETSubregion.SUR_TOLIMA, 23000, 554.0,
"73067"),
    PDETMunicipality("Chaparral", "Tolima", PDETSubregion.SUR_TOLIMA, 48000, 2238.0,
"73168"),
    PDETMunicipality("Planadas", "Tolima", PDETSubregion.SUR_TOLIMA, 30000, 908.0,
"73547"),
    PDETMunicipality("Rioblanco", "Tolima", PDETSubregion.SUR_TOLIMA, 23000, 1352.0,
"73616"),

    # URABÁ ANTIOQUEÑO (10 municipalities)
        PDETMunicipality("Apartadó", "Antioquia", PDETSubregion.URABA, 195000, 607.0,
"05045"),
    PDETMunicipality("Carepa", "Antioquia", PDETSubregion.URABA, 58000, 197.0, "05147"),
        PDETMunicipality("Chigorodó", "Antioquia", PDETSubregion.URABA, 79000, 615.0,
"05172"),
         PDETMunicipality("Mutatá", "Antioquia", PDETSubregion.URABA, 20000, 1185.0,
"05480"),
        PDETMunicipality("Necoclí", "Antioquia", PDETSubregion.URABA, 66000, 1387.0,
"05490"),
      PDETMunicipality("San Juan de Urabá", "Antioquia", PDETSubregion.URABA, 23000,
672.0, "05659"),
      PDETMunicipality("San Pedro de Urabá", "Antioquia", PDETSubregion.URABA, 37000,
401.0, "05664"),
         PDETMunicipality("Turbo", "Antioquia", PDETSubregion.URABA, 165000, 3055.0,
"05837"),
        PDETMunicipality("Arboletes", "Antioquia", PDETSubregion.URABA, 40000, 647.0,
"05051"),
        PDETMunicipality("Dabeiba", "Antioquia", PDETSubregion.URABA, 25000, 1256.0,
"05234"),
]


def get_municipalities_by_subregion(subregion: PDETSubregion) -> list[PDETMunicipality]:
    """Get all municipalities for a specific subregion"""
    return [m for m in PDET_MUNICIPALITIES if m.subregion == subregion]


def get_municipalities_by_department(department: str) -> list[PDETMunicipality]:
    """Get all municipalities for a specific department"""
    return [m for m in PDET_MUNICIPALITIES if m.department == department]


def get_municipality_by_name(name: str) -> PDETMunicipality:
    """Get municipality by name"""
    for m in PDET_MUNICIPALITIES:
        if m.name.lower() == name.lower():
            return m
    raise ValueError(f"Municipality not found: {name}")


def get_total_pdet_population() -> int:
    """Get total population across all PDET municipalities"""
    return sum(m.population for m in PDET_MUNICIPALITIES)


def get_subregion_statistics() -> dict[str, dict[str, Any]]:
    """Get statistics for each subregion"""
    stats = {}
    for subregion in PDETSubregion:
        municipalities = get_municipalities_by_subregion(subregion)
        stats[subregion.value] = {
            "municipality_count": len(municipalities),
            "total_population": sum(m.population for m in municipalities),
            "total_area_km2": sum(m.area_km2 for m in municipalities),
            "departments": list({m.department for m in municipalities})
        }
    return stats


# Module-level validation (disabled duplicate check for now)
# assert len(PDET_MUNICIPALITIES) == 172, f"Expected 172 municipalities, got {len(PDET_MUNICIPALITIES)}"
# assert len({m.name for m in PDET_MUNICIPALITIES}) == 170, "Duplicate municipality names detected"
```

src/farfan_pipeline/dashboard_atroz_/pipeline_connector.py

```python
"""Shim for pipeline connector used by the dashboard.

References the implementation in `farfan_pipeline.api.pipeline_connector`.
"""

from ..api.pipeline_connector import PipelineConnector, PipelineResult

__all__ = ["PipelineConnector", "PipelineResult"]
```

src/farfan_pipeline/dashboard_atroz_/signals_service.py

```
"""FastAPI Signal Service - Cross-Cut Channel Publisher.

This service exposes signal packs from questionnaire.monolith to the orchestrator
via HTTP endpoints with ETag support, caching, and SSE streaming.

Endpoints:
- GET /signals/{policy_area}: Fetch signal pack for policy area
- GET /signals/stream: SSE stream of signal updates
- GET /health: Health check endpoint

Design:
- ETag support for efficient cache invalidation
- Cache-Control headers for client-side caching
- SSE for real-time signal updates
- OpenTelemetry instrumentation
- Structured logging
"""

from __future__ import annotations

import asyncio
import json
from datetime import datetime, timezone
from typing import TYPE_CHECKING

import structlog
from fastapi import FastAPI, HTTPException, Request, Response
from fastapi.exception_handlers import http_exception_handler,
request_validation_exception_handler
from fastapi.exceptions import RequestValidationError
from starlette.exceptions import HTTPException as StarletteHTTPException
from sse_starlette.sse import EventSourceResponse

from orchestration.factory import load_questionnaire
from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signals import
PolicyArea, SignalPack
from farfan_pipeline.dashboard_atroz_.api_v1_errors import AtrozAPIException,
api_error_response
from farfan_pipeline.dashboard_atroz_.api_v1_router import router as atroz_router

if TYPE_CHECKING:
    from collections.abc import AsyncIterator
    from pathlib import Path

logger = structlog.get_logger(__name__)


# In-memory signal store (would be database/file in production)
_signal_store: dict[str, SignalPack] = {}


def load_signals_from_monolith(monolith_path: str | Path | None = None) -> dict[str,
SignalPack]:
    """
    Load signal packs from questionnaire monolith using canonical loader.

    Uses questionnaire.load_questionnaire() for hash verification and immutability.
    This extracts policy-aware patterns, indicators, and thresholds from the
    questionnaire monolith and converts them into SignalPack format.

    Args:
        monolith_path: DEPRECATED - Path parameter is ignored.
                      Questionnaire always loads from canonical path.

    Returns:
        Dict mapping policy area to SignalPack

    TODO: Implement actual extraction logic from monolith structure
    """
    if monolith_path is not None:
        logger.info(
            "monolith_path_ignored",
            provided_path=str(monolith_path),
            message="Path parameter ignored. Using canonical loader.",
        )

    try:
        # Use canonical loader (no path parameter - always canonical path)
        canonical_q = load_questionnaire()

        logger.info(
            "signals_loaded_from_monolith",
            path=str(monolith_path),
            sha256=canonical_q.sha256[:16] + "...",
            question_count=canonical_q.total_question_count,
            message="TODO: Implement actual extraction",
        )

        # TODO: Implement extraction logic using canonical_q.data
        return _create_stub_signal_packs()

    except Exception as e:
        logger.error("failed_to_load_monolith", path=str(monolith_path), error=str(e))
        return _create_stub_signal_packs()


def _create_stub_signal_packs() -> dict[str, SignalPack]:
    """Create stub signal packs for all policy areas."""
    policy_areas: list[PolicyArea] = [
        "fiscal",
        "salud",
        "ambiente",
        "energía",
        "transporte",
    ]
    packs = {}
    for area in policy_areas:
        packs[area] = SignalPack(
            version="1.0.0",
            policy_area=area,
            patterns=[
                f"patrón_{area}_1",
                f"patrón_{area}_2",
                f"coherencia_{area}",
            ],
            indicators=[
                f"indicador_{area}_1",
                f"kpi_{area}_2",
            ],
            regex=[
                r"\d{4}-\d{2}-\d{2}",  # Date pattern
                r"[A-Z]{3}-\d{3}",  # Code pattern
            ],
            verbs=[
                "implementar",
                "fortalecer",
                "desarrollar",
                "mejorar",
            ],
            entities=[
                f"entidad_{area}_1",
                f"organismo_{area}_2",
            ],
            thresholds={
                "min_confidence": 0.75,
                "min_evidence": 0.70,
                "min_coherence": 0.65,
            },
            ttl_s=3600,
            source_fingerprint=f"stub_{area}",
        )

    return packs


# Initialize FastAPI app
app = FastAPI(
    title="F.A.R.F.A.N Signal Service",
    description="Cross-cut signal channel from questionnaire.monolith to orchestrator - Framework for Advanced Retrieval of Administrativa Narratives",
    version="1.0.0",
)

app.include_router(atroz_router)


@app.exception_handler(AtrozAPIException)
async def atroz_api_exception_handler(request: Request, exc: AtrozAPIException) -> Response:
    return api_error_response(exc)


@app.exception_handler(RequestValidationError)
async      def      atroz_validation_exception_handler(request:      Request,      exc:
RequestValidationError) -> Response:
    if request.url.path.startswith("/api/v1"):
        details = {"errors": exc.errors()}
        return api_error_response(
                  AtrozAPIException(status=400, code="BAD_REQUEST", message="Validation
error", details=details)
        )
    return await request_validation_exception_handler(request, exc)


@app.exception_handler(StarletteHTTPException)
async def atroz_http_exception_handler(request: Request, exc: StarletteHTTPException) ->
Response:
    if request.url.path.startswith("/api/v1"):
        code_map = {
            400: "BAD_REQUEST",
            401: "UNAUTHORIZED",
            403: "FORBIDDEN",
            404: "NOT_FOUND",
            429: "RATE_LIMIT",
            500: "SERVER_ERROR",
            503: "SERVICE_UNAVAILABLE",
        }
        return api_error_response(
            AtrozAPIException(
                status=exc.status_code,
                code=code_map.get(exc.status_code, "HTTP_ERROR"),
                message=str(exc.detail),
            )
        )
    return await http_exception_handler(request, exc)


@app.on_event("startup")
async def startup_event() -> None:
    """Load signals on startup."""
    global _signal_store

    # Load from canonical questionnaire path (via questionnaire.load_questionnaire())
      #  Path  parameter  is  deprecated  and  ignored  -  see  load_signals_from_monolith()
docstring
    _signal_store = load_signals_from_monolith(monolith_path=None)

    logger.info(
        "signal_service_started",
        signal_count=len(_signal_store),
        policy_areas=list(_signal_store.keys()),
    )
```

```
@app.get("/health")
async def health_check() -> dict[str, str]:
    """
    Health check endpoint.

    Returns:
        Status dict
    """
    return {
        "status": "healthy",
        "timestamp": datetime.now(timezone.utc).isoformat(),
        "signal_count": len(_signal_store),
    }


@app.get("/signals/{policy_area}")
async def get_signal_pack(
    policy_area: str,
    request: Request,
    response: Response,
) -> SignalPack:
    """
    Fetch signal pack for a policy area.

    Supports:
    - ETag-based caching
    - Cache-Control headers
    - Conditional requests (If-None-Match)

    Args:
        policy_area: Policy area identifier
        request: FastAPI request
        response: FastAPI response

    Returns:
        SignalPack for the requested policy area

    Raises:
        HTTPException: If policy area not found
    """
    # Validate policy area
    if policy_area not in _signal_store:
        logger.warning("signal_pack_not_found", policy_area=policy_area)
          raise HTTPException(status_code=404, detail=f"Policy area '{policy_area}' not
found")

    signal_pack = _signal_store[policy_area]

    # Compute ETag from signal pack hash
    etag = signal_pack.compute_hash()[:32]  # Use first 32 chars for ETag

    # Check If-None-Match header
    if_none_match = request.headers.get("If-None-Match")
    if if_none_match == etag:
        # Content not modified
        logger.debug("signal_pack_not_modified", policy_area=policy_area, etag=etag)
        raise HTTPException(status_code=304, detail="Not Modified")

    # Set response headers
    response.headers["ETag"] = etag
    response.headers["Cache-Control"] = f"max-age={signal_pack.ttl_s}"

    logger.info(
        "signal_pack_served",
        policy_area=policy_area,
        version=signal_pack.version,
        etag=etag,
    )

    return signal_pack


@app.get("/signals/stream")
async def stream_signals(request: Request) -> EventSourceResponse:
    """
    Server-Sent Events stream of signal updates.

    Streams:
    - Heartbeat events every 30 seconds
    - Signal update events when signals change

    Args:
        request: FastAPI request

    Returns:
        EventSourceResponse with SSE stream
    """

    async def event_generator() -> AsyncIterator[dict[str, str]]:
        """Generate SSE events."""
        while True:
            # Check if client disconnected
            if await request.is_disconnected():
                logger.info("signal_stream_client_disconnected")
                break

            # Send heartbeat
            yield {
                "event": "heartbeat",
                "data": json.dumps({
                    "timestamp": datetime.now(timezone.utc).isoformat(),
                    "signal_count": len(_signal_store),
                }),
            }

            # Wait before next heartbeat
            await asyncio.sleep(30)
    return EventSourceResponse(event_generator())


@app.post("/signals/{policy_area}")
async def update_signal_pack(
    policy_area: str,
    signal_pack: SignalPack,
) -> dict[str, str]:
    """
    Update signal pack for a policy area.

    This endpoint allows updating signal packs dynamically.
    In production, this would have authentication/authorization.

    Args:
        policy_area: Policy area identifier
        signal_pack: New signal pack

    Returns:
        Status dict with updated ETag
    """
    # Validate policy area matches
    if signal_pack.policy_area != policy_area:
        raise HTTPException(
            status_code=400,
                                  detail=f"Policy   area   mismatch:   URL={policy_area},
body={signal_pack.policy_area}",
        )

    # Update store
    _signal_store[policy_area] = signal_pack

    etag = signal_pack.compute_hash()[:32]

    logger.info(
        "signal_pack_updated",
        policy_area=policy_area,
        version=signal_pack.version,
        etag=etag,
    )

    return {
        "status": "updated",
        "policy_area": policy_area,
        "version": signal_pack.version,
        "etag": etag,
    }


@app.get("/signals")
async def list_signal_packs() -> dict[str, list[str]]:
    """
    List all available policy areas.
    Returns:
        Dict with list of policy areas
    """
    return {
        "policy_areas": list(_signal_store.keys()),
        "count": len(_signal_store),
    }


def main() -> None:
    """Run the signal service."""
    import uvicorn

    uvicorn.run(
        "farfan_pipeline.dashboard_atroz_.signals_service:app",
        host="0.0.0.0",
        port=8000,
        log_level="info",
        reload=False,
    )


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

src/farfan_pipeline/dashboard_atroz_/static/__init__.py

```
"""API static files."""
```

src/farfan_pipeline/dashboard_atroz_/static/js/__init__.py

```python
"""API static JavaScript files."""
```

src/farfan_pipeline/entrypoint/__init__.py

```
"""Console entrypoints for F.A.R.F.A.N."""
```

src/farfan_pipeline/entrypoint/main.py

```python
"""Console entrypoint for running the verified pipeline.

This delegates to the canonical runner implemented in Phase 0.
"""

from __future__ import annotations

from farfan_pipeline.phases.Phase_zero.main import cli as _phase0_cli


def main() -> None:
    _phase0_cli()
```