```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 7
================================================================================
Generated: 2025-12-07T06:17:17.816218
Files in this batch: 17
================================================================================


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

"""API layer for SAAAAAA system."""



================================================================================
FILE: src/farfan_pipeline/api/api_server.py
================================================================================

#!/usr/bin/env python3
"""AtroZ Dashboard API server with live pipeline integration."""

        self.jobs_dir.mkdir(parents=True, exist_ok=True)
        self.dashboard_transformer = DashboardDataService(self.jobs_dir)

        self.state_lock = Lock()
        self.dashboard_state = self._load_dashboard_state()
        self.region_catalog = self._build_region_catalog()

        self.metrics = {
            'documents_processed': 0,
            'total_questions': 0,
            'total_evidence': 0,
            'total_recommendations': 0,
            'avg_macro_score': None,
            'last_run': None,
            'pipeline_runs': [],
        }

    @calibrated_method("farfan_pipeline.api.api_server.DataService.get_pdet_regions")
    def get_pdet_regions(self) -> list[dict[str, Any]]:
        """Retrieve all PDET regions with live pipeline data from dashboard state."""
        with self.state_lock:
            region_records = [dict(record) for record in self.dashboard_state.get('regions', [])]

        total_regions = max(len(region_records), 1)
        summaries: list[dict[str, Any]] = []
        for index, record in enumerate(region_records):
            coordinates = record.get('coordinates')
            if not coordinates:
                record['coordinates'] = self._compute_region_coordinates(index, total_regions)
            summary, _ = self.dashboard_transformer.summarize_region(record)
            summaries.append(summary)

        return summaries
```

```
    @staticmethod
    def _build_region_catalog() -> dict[str, dict[str, Any]]:
        """Build catalog of PDET region metadata from official statistics."""
        catalog: dict[str, dict[str, Any]] = {}
        try:
            stats = get_subregion_statistics()
        except Exception as exc:
            logger.warning(f"Failed to load PDET statistics: {exc}")
            return catalog

        for name, info in stats.items():
            if not isinstance(info, dict):
                continue
            slug = _slugify_region_name(name)
            catalog[slug] = {
                'name': name,
                'municipalities': info.get('municipality_count', 0),
                'population': info.get('total_population', 0),
                'area': info.get('total_area_km2', 0),
                'departments': info.get('departments', []),
            }
        return catalog

    @staticmethod
    def _to_percentage(value: float | int | None, precision: int = 2) -> float | None:
        """Convert normalized scores (0-1) to percentage values."""
        if value is None:
            return None
        try:
            return round(float(value) * 100.0, precision)
        except (TypeError, ValueError):
            return None

    @staticmethod
    def _extract_macro_score(result: PipelineResult, report_payload: dict[str, Any]) -> float | None:
        """Determine macro score from pipeline result or report payload."""
        if result.macro_score is not None:
            return float(result.macro_score)

        macro_section = report_payload.get('macro_analysis')
        if isinstance(macro_section, dict):
            for key in ('overall_score', 'macro_score', 'adjusted_score', 'score'):
                value = macro_section.get(key)
                if isinstance(value, (int, float)):
                    return float(value)
                if isinstance(value, dict):
                    candidate = value.get('score') or value.get('value')
                    if isinstance(candidate, (int, float)):
                        return float(candidate)
        return None

    @staticmethod
    def _normalize_cluster_scores(raw_clusters: Any) -> dict[str, float]:
        """Normalize cluster structures into a simple slug->score mapping."""
        clusters: dict[str, float] = {}
```

```
113:
114:         if isinstance(raw_clusters, dict):
115:             for key, value in raw_clusters.items():
116:                 score = None
117:                 if isinstance(value, dict):
118:                     score = value.get('score') or value.get('adjusted_score')
119:                 elif isinstance(value, (int, float)):
120:                     score = value
121:                 if isinstance(score, (int, float)):
122:                     slug = _slugify_region_name(str(key))
123:                     clusters[slug] = float(score)
124:                     clusters[str(key).lower()] = float(score)
125:
126:         elif isinstance(raw_clusters, list):
127:             for item in raw_clusters:
128:                 if not isinstance(item, dict):
129:                     continue
130:                 score = item.get('score') or item.get('adjusted_score')
131:                 if not isinstance(score, (int, float)):
132:                     continue
133:                 identifiers = [
134:                     item.get('cluster_id'),
135:                     item.get('id'),
136:                     item.get('name'),
137:                     item.get('label'),
138:                 ]
139:                 for identifier in identifiers:
140:                     if not identifier:
141:                         continue
142:                     slug = _slugify_region_name(str(identifier))
143:                     clusters[slug] = float(score)
144:                     clusters[str(identifier).lower()] = float(score)
145:
146:         return clusters
147:
148:     @staticmethod
149:     def _extract_cluster_score(
150:         normalized_clusters: dict[str, float],
151:         candidates: list[str],
152:     ) -> float | None:
153:         """Pick the first matching cluster score using candidate identifiers."""
154:         for candidate in candidates:
155:             slug = _slugify_region_name(candidate)
156:             if slug in normalized_clusters:
157:                 return normalized_clusters[slug]
158:             lower_key = candidate.lower()
159:             if lower_key in normalized_clusters:
160:                 return normalized_clusters[lower_key]
161:         return None
162:
163:     @staticmethod
164:     def _compute_region_coordinates(index: int, total: int) -> dict[str, float]:
165:         """Place regions on a deterministic circle layout for the dashboard."""
166:         if total <= 0:
167:             return {'x': 50.0, 'y': 50.0}
168:
```

```
        radius = 35.0
        angle = (2 * math.pi * index) / max(total, 1)
        return {
            'x': 50.0 + radius * math.cos(angle),
            'y': 50.0 + radius * math.sin(angle),
        }

    # ========================================================================
    # Dashboard State Management
    # ========================================================================

    def _load_dashboard_state(self) -> dict[str, Any]:
        """Load persisted dashboard state from disk."""
        if self.state_file.exists():
            try:
                with open(self.state_file, 'r', encoding='utf-8') as handle:
                    state = json.load(handle)
                    state.setdefault('regions', [])
                    state.setdefault('evidence', [])
                    return state
            except Exception as exc:
                logger.warning(f"Failed to load dashboard state, resetting: {exc}")
        return {'regions': [], 'evidence': [], 'metrics': {}}

    def _persist_dashboard_state(self) -> None:
        """Persist dashboard state atomically."""
        tmp_path = self.state_file.with_suffix('.tmp')
        with open(tmp_path, 'w', encoding='utf-8') as handle:
            json.dump(self.dashboard_state, handle, indent=2, ensure_ascii=False)
        tmp_path.replace(self.state_file)

    def _store_job_artifact(self, job_id: str, payload: dict[str, Any]) -> Path:
        """Persist raw pipeline output for later retrieval."""
        job_path = self.jobs_dir / f"{job_id}.json"
        with open(job_path, 'w', encoding='utf-8') as handle:
            json.dump(payload, handle, indent=2, ensure_ascii=False)
        return job_path

    def _update_dashboard_state(
        self,
        job_id: str,
        result: PipelineResult,
        report_payload: dict[str, Any],
    ) -> None:
        """Merge pipeline results into dashboard state."""
        timestamp = datetime.now().isoformat()
        municipality = result.metadata.get('municipality', result.document_id)
        payload = report_payload or {}
        report_path = str(self.jobs_dir / f"{job_id}.json")

        region_name = next(
            (
                candidate
                for candidate in (
                    payload.get('region_name'),
                    payload.get('metadata', {}).get('region_name') if isinstance(payload.get('metadata'), dict) else None,
```

```
                    result.metadata.get('pdet_region'),
                    result.metadata.get('region'),
                    municipality,
                )
                if candidate
            ),
            municipality,
        )
        region_slug = _slugify_region_name(region_name)

        macro_score_raw = self._extract_macro_score(result, payload)
        cluster_payload = result.meso_scores
        if not cluster_payload:
            meso_section = payload.get('meso_analysis')
            if isinstance(meso_section, dict):
                cluster_payload = meso_section.get('cluster_scores')
        normalized_clusters = self._normalize_cluster_scores(cluster_payload)

        governance_raw = self._extract_cluster_score(normalized_clusters, ['gobernanza', 'governance', 'cl01'])
        social_raw = self._extract_cluster_score(normalized_clusters, ['social', 'cl02'])
        economic_raw = self._extract_cluster_score(normalized_clusters, ['economico', 'econÃ³mico', 'economic', 'cl03'])
        environmental_raw = self._extract_cluster_score(normalized_clusters, ['ambiental', 'environmental', 'cl04'])

        raw_scores = {
            'overall': macro_score_raw,
            'governance': governance_raw,
            'social': social_raw,
            'economic': economic_raw,
            'environmental': environmental_raw,
        }
        percent_scores = {key: self._to_percentage(value) for key, value in raw_scores.items()}
        cluster_scores_percent = {
            key: self._to_percentage(value) for key, value in normalized_clusters.items()
        }

        macro_band = None
        macro_section = payload.get('macro_analysis')
        if isinstance(macro_section, dict):
            macro_band = macro_section.get('quality_band') or macro_section.get('macro_band')
        if macro_band is None:
            macro_band = result.metadata.get('macro_band')

        region_stats_ref = self.region_catalog.get(region_slug, {})
        stats_payload = {
            'municipalities': region_stats_ref.get('municipalities', 0),
            'population': region_stats_ref.get('population', 0),
            'area': region_stats_ref.get('area', 0),
            'departments': region_stats_ref.get('departments', []),
        }

        indicators_payload = {
            'alignment': percent_scores.get('overall'),
            'implementation': percent_scores.get('governance'),
            'impact': percent_scores.get('environmental') or percent_scores.get('social'),
        }

```

```
        with self.state_lock:
            regions = self.dashboard_state.setdefault('regions', [])
            region_record = next((r for r in regions if r.get('id') == region_slug), None)

            if region_record is None:
                coords = self._compute_region_coordinates(len(regions), max(1, len(regions) + 1))
                region_record = {
                    'id': region_slug,
                    'job_id': job_id,
                    'name': region_name.upper(),
                    'municipality': municipality,
                    'coordinates': coords,
                    'stats': stats_payload,
                    'raw_scores': raw_scores,
                    'scores': percent_scores,
                    'cluster_scores_raw': normalized_clusters,
                    'cluster_scores': cluster_scores_percent,
                    'indicators': indicators_payload,
                    'macro_band': macro_band,
                    'connections': list(region_stats_ref.get('connections', [])),
                    'updated_at': timestamp,
                    'latest_report': job_id,
                    'report_path': report_path,
                }
                regions.append(region_record)
            else:
                region_record['job_id'] = job_id
                region_record['latest_report'] = job_id
                region_record['name'] = region_name.upper()
                region_record['municipality'] = municipality
                region_record['stats'] = stats_payload or region_record.get('stats', {})
                region_record.setdefault('coordinates', self._compute_region_coordinates(0, max(1, len(regions))))
                region_record['raw_scores'] = raw_scores
                region_record['scores'] = percent_scores
                region_record['cluster_scores_raw'] = normalized_clusters
                region_record['cluster_scores'] = cluster_scores_percent
                region_record['indicators'] = indicators_payload
                if not region_record.get('connections'):
                    region_record['connections'] = list(region_stats_ref.get('connections', []))
                region_record['macro_band'] = macro_band
                region_record['updated_at'] = timestamp
                region_record['report_path'] = report_path

            evidence_items = self.dashboard_state.setdefault('evidence', [])
            micro = payload.get('micro_analysis', {})
            questions = micro.get('questions', [])
            for question in questions:
                for evidence in question.get('evidence', []):
                    evidence_items.append(
                        {
                            'source': evidence.get('source', 'Desconocido'),
                            'page': evidence.get('page', '?'),
                            'text': evidence.get('excerpt', ''),
                            'timestamp': timestamp,
                            'job_id': job_id,
                            'region_id': region_slug,
```

```
                        }
                    )
            evidence_items[:] = evidence_items[-200:]

            metrics = self.dashboard_state.setdefault('metrics', self.metrics)
            metrics['documents_processed'] = metrics.get('documents_processed', 0) + 1
            metrics['total_questions'] = metrics.get('total_questions', 0) + result.questions_analyzed
            metrics['total_evidence'] = metrics.get('total_evidence', 0) + result.evidence_count
            metrics['total_recommendations'] = metrics.get('total_recommendations', 0) + result.recommendations_count
            metrics['last_run'] = timestamp
            if macro_score_raw is not None:
                runs = metrics.get('pipeline_runs', [])
                runs.append(macro_score_raw)
                metrics['pipeline_runs'] = runs
                metrics['avg_macro_score'] = sum(runs) / len(runs) if runs else None

            self._persist_dashboard_state()

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

    def start_pipeline_job(
        self,
        pdf_path: Path,
        municipality: str,
        settings: dict[str, Any] | None = None,
    ) -> str:
        """Kick off background pipeline execution for a PDF."""
        job_id = f"job-{uuid.uuid4().hex[:12]}"
        socketio.start_background_task(
            self._run_pipeline_job,
            job_id,
            Path(pdf_path),
            municipality,
            settings or {},
        )
        return job_id

    def _run_pipeline_job(
        self,
        job_id: str,
        pdf_path: Path,
        municipality: str,
        settings: dict[str, Any],
    ) -> None:
        """Execute pipeline in background thread and emit progress events."""

        def progress_callback(phase: int, phase_name: str) -> None:
            status = self.pipeline_connector.get_job_status(job_id) or {}
            socketio.emit(
                'analysis_progress',
                {
                    'job_id': job_id,
                    'phase': phase_name,
                    'phase_num': phase,
```

```
                    'progress': status.get('progress', 0),
                },
            )

        loop = asyncio.new_event_loop()
        try:
            asyncio.set_event_loop(loop)
            result = loop.run_until_complete(
                self.pipeline_connector.execute_pipeline(
                    pdf_path=str(pdf_path),
                    job_id=job_id,
                    municipality=municipality,
                    progress_callback=progress_callback,
                    settings=settings,
                )
            )
            report = {}
            output_path = result.metadata.get('output_path')
            if output_path and Path(output_path).exists():
                with open(output_path, 'r', encoding='utf-8') as handle:
                    report = json.load(handle)
            artifact_path = self._store_job_artifact(
                job_id,
                {
                    'pipeline_result': result.__dict__,
                    'report': report,
                },
            )
            self._update_dashboard_state(job_id, result, report or {})
            socketio.emit(
                'analysis_complete',
                {
                    'job_id': job_id,
                    'result': {
                        'macro_score': result.macro_score,
                        'meso_scores': result.meso_scores,
                        'micro_scores': result.micro_scores,
                        'report_path': str(artifact_path),
                    },
                },
            )
        except Exception as exc:
            logger.error(f"Pipeline job {job_id} failed: {exc}", exc_info=True)
            socketio.emit(
                'analysis_error',
                {
                    'job_id': job_id,
                    'error': str(exc),
                },
            )
        finally:
            asyncio.set_event_loop(None)
            loop.close()

    def get_job_status(self, job_id: str) -> dict[str, Any] | None:
        """Expose connector job status."""
```

```
449:         return self.pipeline_connector.get_job_status(job_id)
450:
451:     def get_metrics_snapshot(self) -> dict[str, Any]:
452:         """Return cached metrics for admin dashboard."""
453:         with self.state_lock:
454:             metrics = dict(self.dashboard_state.get('metrics', {}))
455:             runs = metrics.pop('pipeline_runs', [])
456:             metrics['run_count'] = len(runs)
457:             return metrics
458:
459:     @calibrated_method("farfan_core.api.api_server.DataService.get_pdet_regions")
460:     def get_pdet_regions(self) -> list[dict[str, Any]]:
461:         """
462:         Get all PDET regions with scores
463:
464:         Returns data in format expected by AtroZ dashboard
465:         """
466:         # PDET regions from Colombian government definition
467:         regions = [
468:             {
469:                 'id': 'alto-patia',
470:                 'name': 'ALTO PATÃ\215A Y NORTE DEL CAUCA',
471:                 'coordinates': {'x': 25, 'y': 20},
472:                 'metadata': {
473:                     'municipalities': 24,
474:                     'population': 450000,
475:                     'area': 12500
476:                 },
477:                 'scores': {
478:                     'overall': 72,
479:                     'governance': 68,
480:                     'social': 74,
481:                     'economic': 70,
482:                     'environmental': 75,
483:                     'lastUpdated': datetime.now().isoformat()
484:                 },
485:                 'connections': ['pacifico-medio', 'sur-tolima'],
486:                 'indicators': {
487:                     'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L277_33", 0.72),
488:                     'implementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L278_38", 0.68),
489:                     'impact': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L279_30", 0.75)
490:                 }
491:             },
492:             {
493:                 'id': 'arauca',
494:                 'name': 'ARAUCA',
495:                 'coordinates': {'x': 75, 'y': 15},
496:                 'metadata': {
497:                     'municipalities': 4,
498:                     'population': 95000,
499:                     'area': 23818
500:                 },
501:                 'scores': {
502:                     'overall': 68,
503:                     'governance': 65,
504:                     'social': 70,
```

```
                    'economic': 67,
                    'environmental': 71,
                    'lastUpdated': datetime.now().isoformat()
                },
                'connections': ['catatumbo'],
                'indicators': {
                    'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L301_33", 0.68),
                    'implementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L302_38", 0.65),
                    'impact': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L303_30", 0.70)
                }
            },
            {
                'id': 'bajo-cauca',
                'name': 'BAJO CAUCA Y NORDESTE ANTIOQUEÃ\221O',
                'coordinates': {'x': 45, 'y': 25},
                'metadata': {'municipalities': 13, 'population': 280000, 'area': 8485},
                'scores': {'overall': 65, 'governance': 62, 'social': 66, 'economic': 64, 'environmental': 68, 'lastUpdated': datetime.now().isoformat()},
                'connections': ['sur-cordoba', 'sur-bolivar'],
                'indicators': {'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L313_44", 0.65), 'i
mplementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L313_68", 0.62), 'impact': ParameterLoaderV2.get("farf
an_core.api.api_server.DataService.get_pdet_regions", "auto_param_L313_84", 0.67)}
            },
            {
                'id': 'catatumbo',
                'name': 'CATATUMBO',
                'coordinates': {'x': 65, 'y': 20},
                'metadata': {'municipalities': 11, 'population': 220000, 'area': 11700},
                'scores': {'overall': 61, 'governance': 58, 'social': 62, 'economic': 60, 'environmental': 64, 'lastUpdated': datetime.now().isoformat()},
                'connections': ['arauca'],
                'indicators': {'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L322_44", 0.61), 'i
mplementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L322_68", 0.58), 'impact': ParameterLoaderV2.get("farf
an_core.api.api_server.DataService.get_pdet_regions", "auto_param_L322_84", 0.63)}
            },
            {
                'id': 'choco',
                'name': 'CHOCÃ\223',
                'coordinates': {'x': 15, 'y': 35},
                'metadata': {'municipalities': 14, 'population': 180000, 'area': 43000},
                'scores': {'overall': 58, 'governance': 55, 'social': 59, 'economic': 57, 'environmental': 61, 'lastUpdated': datetime.now().isoformat()},
                'connections': ['uraba', 'pacifico-medio'],
                'indicators': {'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L331_44", 0.58), 'i
mplementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L331_68", 0.55), 'impact': ParameterLoaderV2.get("farf
an_core.api.api_server.DataService.get_pdet_regions", "auto_param_L331_84", 0.60)}
            },
            {
                'id': 'caguan',
                'name': 'CUENCA DEL CAGUÃ\201N Y PIEDEMONTE CAQUETEÃ\221O',
                'coordinates': {'x': 55, 'y': 40},
                'metadata': {'municipalities': 17, 'population': 350000, 'area': 39000},
                'scores': {'overall': 70, 'governance': 67, 'social': 71, 'economic': 69, 'environmental': 72, 'lastUpdated': datetime.now().isoformat()},
                'connections': ['macarena', 'putumayo'],
                'indicators': {'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L340_44", 0.70), 'i
mplementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L340_68", 0.67), 'impact': ParameterLoaderV2.get("farf
an_core.api.api_server.DataService.get_pdet_regions", "auto_param_L340_84", 0.71)}
            },
            {
```

```
  553:                 'id': 'macarena',
  554:                 'name': 'MACARENA-GUAVIARE',
  555:                 'coordinates': {'x': 60, 'y': 55},
  556:                 'metadata': {'municipalities': 10, 'population': 140000, 'area': 32000},
  557:                 'scores': {'overall': 66, 'governance': 63, 'social': 67, 'economic': 65, 'environmental': 68, 'lastUpdated': datetime.now().isoformat()},
  558:                 'connections': ['caguan'],
  559:                 'indicators': {'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L349_44", 0.66), 'i
mplementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L349_68", 0.63), 'impact': ParameterLoaderV2.get("farf
an_core.api.api_server.DataService.get_pdet_regions", "auto_param_L349_84", 0.67)}
  560:             },
  561:             {
  562:                 'id': 'montes-maria',
  563:                 'name': 'MONTES DE MARÃ\215A',
  564:                 'coordinates': {'x': 40, 'y': 10},
  565:                 'metadata': {'municipalities': 15, 'population': 330000, 'area': 6500},
  566:                 'scores': {'overall': 74, 'governance': 71, 'social': 75, 'economic': 73, 'environmental': 76, 'lastUpdated': datetime.now().isoformat()},
  567:                 'connections': ['sur-bolivar'],
  568:                 'indicators': {'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L358_44", 0.74), 'i
mplementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L358_68", 0.71), 'impact': ParameterLoaderV2.get("farf
an_core.api.api_server.DataService.get_pdet_regions", "auto_param_L358_84", 0.75)}
  569:             },
  570:             {
  571:                 'id': 'pacifico-medio',
  572:                 'name': 'PACÃ\215FICO MEDIO',
  573:                 'coordinates': {'x': 10, 'y': 50},
  574:                 'metadata': {'municipalities': 4, 'population': 120000, 'area': 10000},
  575:                 'scores': {'overall': 62, 'governance': 59, 'social': 63, 'economic': 61, 'environmental': 64, 'lastUpdated': datetime.now().isoformat()},
  576:                 'connections': ['choco', 'alto-patia'],
  577:                 'indicators': {'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L367_44", 0.62), 'i
mplementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L367_68", 0.59), 'impact': ParameterLoaderV2.get("farf
an_core.api.api_server.DataService.get_pdet_regions", "auto_param_L367_84", 0.63)}
  578:             },
  579:             {
  580:                 'id': 'pacifico-narinense',
  581:                 'name': 'PACÃ\215FICO Y FRONTERA NARIÃ\221ENSE',
  582:                 'coordinates': {'x': 5, 'y': 65},
  583:                 'metadata': {'municipalities': 11, 'population': 190000, 'area': 14000},
  584:                 'scores': {'overall': 59, 'governance': 56, 'social': 60, 'economic': 58, 'environmental': 61, 'lastUpdated': datetime.now().isoformat()},
  585:                 'connections': ['putumayo'],
  586:                 'indicators': {'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L376_44", 0.59), 'i
mplementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L376_68", 0.56), 'impact': ParameterLoaderV2.get("farf
an_core.api.api_server.DataService.get_pdet_regions", "auto_param_L376_84", 0.60)}
  587:             },
  588:             {
  589:                 'id': 'putumayo',
  590:                 'name': 'PUTUMAYO',
  591:                 'coordinates': {'x': 35, 'y': 70},
  592:                 'metadata': {'municipalities': 11, 'population': 270000, 'area': 25000},
  593:                 'scores': {'overall': 67, 'governance': 64, 'social': 68, 'economic': 66, 'environmental': 69, 'lastUpdated': datetime.now().isoformat()},
  594:                 'connections': ['caguan', 'pacifico-narinense'],
  595:                 'indicators': {'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L385_44", 0.67), 'i
mplementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L385_68", 0.64), 'impact': ParameterLoaderV2.get("farf
an_core.api.api_server.DataService.get_pdet_regions", "auto_param_L385_84", 0.68)}
  596:             },
  597:             {
  598:                 'id': 'sierra-nevada',
```

```
  599:                 'name': 'SIERRA NEVADA - PERIJÃ\201 - ZONA BANANERA',
  600:                 'coordinates': {'x': 70, 'y': 5},
  601:                 'metadata': {'municipalities': 10, 'population': 380000, 'area': 15000},
  602:                 'scores': {'overall': 63, 'governance': 60, 'social': 64, 'economic': 62, 'environmental': 65, 'lastUpdated': datetime.now().isoformat()},
  603:                 'connections': ['catatumbo'],
  604:                 'indicators': {'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L394_44", 0.63), 'i
mplementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L394_68", 0.60), 'impact': ParameterLoaderV2.get("farf
an_core.api.api_server.DataService.get_pdet_regions", "auto_param_L394_84", 0.64)}
  605:             },
  606:             {
  607:                 'id': 'sur-bolivar',
  608:                 'name': 'SUR DE BOLÃ\215VAR',
  609:                 'coordinates': {'x': 50, 'y': 15},
  610:                 'metadata': {'municipalities': 7, 'population': 150000, 'area': 7000},
  611:                 'scores': {'overall': 60, 'governance': 57, 'social': 61, 'economic': 59, 'environmental': 62, 'lastUpdated': datetime.now().isoformat()},
  612:                 'connections': ['bajo-cauca', 'montes-maria'],
  613:                 'indicators': {'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L403_44", 0.60), 'i
mplementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L403_68", 0.57), 'impact': ParameterLoaderV2.get("farf
an_core.api.api_server.DataService.get_pdet_regions", "auto_param_L403_84", 0.61)}
  614:             },
  615:             {
  616:                 'id': 'sur-cordoba',
  617:                 'name': 'SUR DE CÃ\223RDOBA',
  618:                 'coordinates': {'x': 35, 'y': 15},
  619:                 'metadata': {'municipalities': 5, 'population': 180000, 'area': 4500},
  620:                 'scores': {'overall': 69, 'governance': 66, 'social': 70, 'economic': 68, 'environmental': 71, 'lastUpdated': datetime.now().isoformat()},
  621:                 'connections': ['bajo-cauca', 'uraba'],
  622:                 'indicators': {'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L412_44", 0.69), 'i
mplementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L412_68", 0.66), 'impact': ParameterLoaderV2.get("farf
an_core.api.api_server.DataService.get_pdet_regions", "auto_param_L412_84", 0.70)}
  623:             },
  624:             {
  625:                 'id': 'sur-tolima',
  626:                 'name': 'SUR DEL TOLIMA',
  627:                 'coordinates': {'x': 45, 'y': 45},
  628:                 'metadata': {'municipalities': 4, 'population': 110000, 'area': 3500},
  629:                 'scores': {'overall': 71, 'governance': 68, 'social': 72, 'economic': 70, 'environmental': 73, 'lastUpdated': datetime.now().isoformat()},
  630:                 'connections': ['alto-patia', 'caguan'],
  631:                 'indicators': {'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L421_44", 0.71), 'i
mplementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L421_68", 0.68), 'impact': ParameterLoaderV2.get("farf
an_core.api.api_server.DataService.get_pdet_regions", "auto_param_L421_84", 0.72)}
  632:             },
  633:             {
  634:                 'id': 'uraba',
  635:                 'name': 'URABÃ\201 ANTIOQUEÃ\221O',
  636:                 'coordinates': {'x': 20, 'y': 10},
  637:                 'metadata': {'municipalities': 10, 'population': 420000, 'area': 11600},
  638:                 'scores': {'overall': 64, 'governance': 61, 'social': 65, 'economic': 63, 'environmental': 66, 'lastUpdated': datetime.now().isoformat()},
  639:                 'connections': ['choco', 'sur-cordoba'],
  640:                 'indicators': {'alignment': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L430_44", 0.64), 'i
mplementation': ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_pdet_regions", "auto_param_L430_68", 0.61), 'impact': ParameterLoaderV2.get("farf
an_core.api.api_server.DataService.get_pdet_regions", "auto_param_L430_84", 0.65)}
  641:             }
  642:         ]
  643:         return regions
  644:
```

```
    def get_constellation_map_data(self) -> dict[str, Any]:
        """Build constellation graph based on region and cluster relationships."""
        with self.state_lock:
            region_records = [dict(record) for record in self.dashboard_state.get('regions', [])]

        nodes: list[dict[str, Any]] = []
        links: list[dict[str, Any]] = []
        cluster_nodes: dict[str, dict[str, Any]] = {}

        for record in region_records:
            summary, context = self.dashboard_transformer.summarize_region(record)
            region_id = summary.get('id')
            if not region_id:
                continue

            nodes.append(
                {
                    'id': region_id,
                    'name': summary.get('name'),
                    'type': 'region',
                    'group': 1,
                    'score': summary.get('scores', {}).get('overall'),
                }
            )

            for cluster in context.get('clusters', []):
                cluster_id = cluster.get('cluster_id')
                if not cluster_id:
                    continue
                cluster_entry = cluster_nodes.setdefault(
                    cluster_id,
                    {
                        'id': cluster_id,
                        'name': cluster.get('name'),
                        'type': 'cluster',
                        'group': 2,
                        'score_total': 0.0,
                        'count': 0,
                    },
                )
                score_percent = cluster.get('score_percent')
                if score_percent is not None:
                    cluster_entry['score_total'] += score_percent
                    cluster_entry['count'] += 1

                links.append(
                    {
                        'source': region_id,
                        'target': cluster_id,
                        'value': score_percent if score_percent is not None else 0.0,
                    }
                )

        for cluster_id, entry in cluster_nodes.items():
            count = entry.pop('count') or 1
            score_total = entry.pop('score_total')
```

```
            entry['score'] = round(score_total / count, 2)
            nodes.append(entry)

        return {'nodes': nodes, 'links': links}

    @calibrated_method("farfan_core.api.api_server.DataService.get_region_detail")
    def get_region_detail(self, region_id: str) -> dict[str, Any] | None:
        """Get detailed region payload with macro/meso/micro layers."""
        with self.state_lock:
            region_record = None
            evidence_items: list[dict[str, Any]] = []
            for record in self.dashboard_state.get('regions', []):
                if record.get('id') == region_id:
                    region_record = dict(record)
                    break
            if region_record:
                evidence_items = [dict(item) for item in self.dashboard_state.get('evidence', []) if item.get('region_id') == region_id]

        if region_record is None:
            return None

        summary, context = self.dashboard_transformer.summarize_region(region_record)
        return self.dashboard_transformer.build_region_detail(region_record, summary, context, evidence_items)


# Initialize data service
data_service = DataService()

# Initialize recommendation engine via factory
recommendation_engine = None
try:
    recommendation_engine = create_recommendation_engine(enable_cache=True)
    logger.info("Recommendation engine initialized successfully via factory")
except Exception as e:
    logger.warning(f"Failed to initialize recommendation engine: {e}")

# Track application start time for uptime metrics
app_start_time = datetime.now()

# ============================================================================
# API ENDPOINTS
# ============================================================================

@app.route('/')
def dashboard():
    """Serve the AtroZ dashboard"""
    from flask import send_from_directory
    return send_from_directory(app.static_folder, 'index.html')

@app.route('/api/v1/health', methods=['GET'])
def health_check():
    """Health check endpoint"""
    return jsonify({
        'status': 'healthy',
        'timestamp': datetime.now().isoformat(),
        'version': 'ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_evidence_stream", "auto_param_L572_20", 1.0).0'
```

```
    })

@app.route('/api/v1/auth/token', methods=['POST'])
@rate_limit
def get_auth_token():
    """Get authentication token"""
    data = request.get_json()
    client_id = data.get('client_id')
    client_secret = data.get('client_secret')

    # Validate credentials (implement proper validation in production)
    if not client_id or not client_secret:
        return jsonify({'error': 'Missing credentials'}), 400

    # Generate token
    token = generate_jwt_token(client_id)

    return jsonify({
        'access_token': token,
        'token_type': 'Bearer',
        'expires_in': APIConfig.JWT_EXPIRATION_HOURS * 3600
    })

@app.route('/api/v1/constellation_map', methods=['GET'])
@rate_limit
@cache_response(timeout=300)
def get_constellation_map():
    """
    Get data for the constellation map visualization

    Returns:
        JSON object with nodes and links for the constellation map
    """
    try:
        constellation_data = data_service.get_constellation_map_data()

        return jsonify({
            'status': 'success',
            'data': constellation_data,
            'timestamp': datetime.now().isoformat()
        })

    except Exception as e:
        logger.error(f"Failed to get constellation map data: {e}")
        return jsonify({'error': str(e)}), 500


@app.route('/api/v1/pdet/regions', methods=['GET'])
@rate_limit
@cache_response(timeout=300)
def get_pdet_regions():
    """
    Get all PDET regions with scores

    Returns:
        List of PDET regions with metadata and scores
```

```
    """
    try:
        regions = data_service.get_pdet_regions()

        return jsonify({
            'status': 'success',
            'data': regions,
            'count': len(regions),
            'timestamp': datetime.now().isoformat()
        })

    except Exception as e:
        logger.error(f"Failed to get PDET regions: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/api/v1/pdet/regions/<region_id>', methods=['GET'])
@rate_limit
@cache_response(timeout=300)
def get_region_detail(region_id: str):
    """
    Get detailed information for a specific PDET region

    Args:
        region_id: Region identifier (e.g., 'alto-patia')

    Returns:
        Detailed region data with analysis
    """
    try:
        region = data_service.get_region_detail(region_id)

        if not region:
            return jsonify({'error': 'Region not found'}), 404

        return jsonify({
            'status': 'success',
            'data': region,
            'timestamp': datetime.now().isoformat()
        })

    except Exception as e:
        logger.error(f"Failed to get region detail: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/api/v1/municipalities/<municipality_id>', methods=['GET'])
@rate_limit
@cache_response(timeout=300)
def get_municipality_data(municipality_id: str):
    """Get REAL municipality data from PDET catalog + pipeline results."""
    from farfan_pipeline.api.pdet_colombia_data import PDET_MUNICIPALITIES, get_municipality_by_name

    try:
        municipality_name = municipality_id.replace('-', ' ').title()

        try:
            municipality = get_municipality_by_name(municipality_name)
```

```
        except ValueError:
            return jsonify({'error': f'Municipality "{municipality_name}" not found'}), 404

        has_analysis = False
        macro_data = None
        meso_data = []
        micro_data = []
        region_slug = _slugify_region_name(municipality.subregion.value)

        with self.state_lock:
            region_state = next(
                (dict(record) for record in self.dashboard_state.get('regions', []) if record.get('id') == region_slug),
                None,
            )

        if region_state:
            has_analysis = True

            summary, context = self.dashboard_transformer.summarize_region(region_state)
            macro_detail = context.get('macro', {})
            clusters = context.get('clusters', [])
            question_matrix = self.dashboard_transformer.extract_question_matrix(context.get('report', {}))

            macro_data = {
                'score': macro_detail.get('score'),
                'score_percent': macro_detail.get('score_percent'),
                'band': macro_detail.get('band') or summary.get('macroBand') or 'SIN DATOS',
                'coherence': macro_detail.get('coherence'),
                'systemic_gaps': macro_detail.get('systemic_gaps', []),
                'alignment': macro_detail.get('alignment'),
            }

            meso_data = [
                {
                    'cluster_id': cluster.get('cluster_id'),
                    'name': cluster.get('name'),
                    'score': cluster.get('score'),
                    'score_percent': cluster.get('score_percent'),
                    'areas': cluster.get('areas'),
                    'weakest_area': cluster.get('weakest_area'),
                    'coherence': cluster.get('coherence'),
                }
                for cluster in clusters
            ]

            micro_data = [
                {
                    'question_id': question.get('id'),
                    'policy_area': question.get('category'),
                    'dimension': question.get('dimension'),
                    'score': question.get('score'),
                    'score_percent': question.get('score_percent'),
                    'evidence': question.get('evidence'),
                    'recommendations': question.get('recommendations'),
                }
                for question in question_matrix
```

```
            ]

        return jsonify({
            'status': 'success',
            'municipality': {
                'id': municipality_id,
                'name': municipality.name,
                'department': municipality.department,
                'dane_code': municipality.dane_code,
                'population': municipality.population,
                'area_km2': municipality.area_km2,
                'subregion': municipality.subregion.value
            },
            'has_analysis': has_analysis,
            'macro': macro_data,
            'meso': meso_data,
            'micro': micro_data,
            'timestamp': datetime.now().isoformat()
        })

    except Exception as e:
        logger.error(f"Failed to get municipality data: {e}")
        return jsonify({'error': str(e)}), 500

    @calibrated_method("farfan_core.api.api_server.DataService.get_evidence_stream")
    def get_evidence_stream(self) -> list[dict[str, Any]]:
        """Return normalized evidence entries for ticker display."""
        with self.state_lock:
            evidence_items = [dict(item) for item in self.dashboard_state.get('evidence', [])]
        return self.dashboard_transformer.normalize_evidence_stream(evidence_items)

@app.route('/api/v1/evidence/stream', methods=['GET'])
@rate_limit
@cache_response(timeout=60)
def get_evidence_stream():
    """
    Get evidence stream for ticker display

    Returns:
        List of evidence items with sources and timestamps
    """
    try:
        evidence = data_service.get_evidence_stream()

        return jsonify({
            'status': 'success',
            'data': evidence,
            'count': len(evidence),
            'timestamp': datetime.now().isoformat()
        })

    except Exception as e:
        logger.error(f"Failed to get evidence stream: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/api/v1/export/dashboard', methods=['POST'])
```

```
@rate_limit
def export_dashboard_data():
    """
    Export dashboard data in various formats

    Request body:
        {
            "format": "json|csv|pdf",
            "regions": ["region_id1", "region_id2"],
            "include_evidence": true
        }

    Returns:
        Exported data file
    """
    try:
        data = request.get_json()
        export_format = data.get('format', 'json')
        region_ids = data.get('regions', [])
        include_evidence = data.get('include_evidence', False)

        # Collect data
        export_data = {
            'timestamp': datetime.now().isoformat(),
            'regions': [],
            'evidence': [] if include_evidence else None
        }

        # Get region data
        for region_id in region_ids:
            region = data_service.get_region_detail(region_id)
            if region:
                export_data['regions'].append(region)

        # Get evidence if requested
        if include_evidence:
            export_data['evidence'] = data_service.get_evidence_stream()

        # Format response based on requested format
        if export_format == 'json':
            return jsonify({
                'status': 'success',
                'data': export_data
            })
        else:
            return jsonify({'error': f'Format {export_format} not yet implemented'}), 400

    except Exception as e:
        logger.error(f"Failed to export dashboard data: {e}")
        return jsonify({'error': str(e)}), 500

# ============================================================================
# ADMIN ENDPOINTS
# ============================================================================

@app.route('/api/admin/metrics', methods=['GET'])
```

```
@cache_response(timeout=2)
def get_admin_metrics():
    """
    Get comprehensive system metrics for admin dashboard.

    Returns all pipeline metrics, orchestrator stats, and performance data
    aligned with AtroZ aesthetic requirements.
    """
    try:
        with data_service.state_lock:
            metrics = data_service.dashboard_state.get('metrics', data_service.metrics)

        pipeline_runs = metrics.get('pipeline_runs', [])

        response_payload = {
            'documents_processed': metrics.get('documents_processed', 0),
            'total_questions': metrics.get('total_questions', 0),
            'total_evidence': metrics.get('total_evidence', 0),
            'total_recommendations': metrics.get('total_recommendations', 0),
            'avg_macro_score': metrics.get('avg_macro_score'),
            'last_run': metrics.get('last_run'),
            'uptime_seconds': int((datetime.now() - app_start_time).total_seconds()) if 'app_start_time' in globals() else 0,
            'calibration_version': ParameterLoaderV2.get('farfan_pipeline.api.api_server', 'calibration_version', '2.0'),
            'method_count': 300,
            'question_count': 300,
            'last_verification': metrics.get('last_run'),
            'perf_macro': metrics.get('perf_macro', 2.3),
            'perf_meso': metrics.get('perf_meso', 1.8),
            'perf_micro': metrics.get('perf_micro', 15.4),
            'perf_report': metrics.get('perf_report', 0.5),
            'estimated_time': 22,
            'pipeline_runs_count': len(pipeline_runs),
        }

        return jsonify(response_payload)

    except Exception as exc:
        logger.error(f"Failed to retrieve admin metrics: {exc}")
        return jsonify({'error': str(exc)}), 500

@app.route('/api/admin/health', methods=['GET'])
@cache_response(timeout=1)
def get_system_health():
    """
    Get real-time system health metrics (CPU, memory, cache, latency).

    Returns resource utilization data for AtroZ health monitor visualization.
    """
    try:
        import psutil

        cpu_percent = psutil.cpu_percent(interval=0.1)
        memory = psutil.virtual_memory()

        cache_hits = getattr(cache_response, 'hits', 0)
        cache_misses = getattr(cache_response, 'misses', 0)
```

```
        cache_total = cache_hits + cache_misses
        cache_hit_rate = (cache_hits / cache_total * 100) if cache_total > 0 else 0

        start_time = datetime.now()
        _ = data_service.get_pdet_regions()
        api_latency = (datetime.now() - start_time).total_seconds() * 1000

        return jsonify({
            'cpu': cpu_percent,
            'memory': memory.percent,
            'cache_hit_rate': cache_hit_rate,
            'api_latency': api_latency,
            'timestamp': datetime.now().isoformat(),
        })

    except ImportError:
        return jsonify({
            'cpu': 0,
            'memory': 0,
            'cache_hit_rate': 0,
            'api_latency': 0,
            'timestamp': datetime.now().isoformat(),
            'note': 'psutil not available',
        })
    except Exception as exc:
        logger.error(f"Failed to retrieve system health: {exc}")
        return jsonify({'error': str(exc)}), 500

@app.route('/api/admin/upload', methods=['POST'])
@require_auth
def upload_pdf_document():
    """
    Upload PDF document for pipeline analysis.

    Accepts multipart/form-data with 'file' field containing PDF.
    Returns document_id for subsequent analysis triggering.
    """
    try:
        if 'file' not in request.files:
            return jsonify({'error': 'No file provided'}), 400

        uploaded = request.files['file']
        if uploaded.filename == '':
            return jsonify({'error': 'Empty filename'}), 400

        if not uploaded.filename.lower().endswith('.pdf'):
            return jsonify({'error': 'Only PDF files accepted'}), 400

        uploads_dir = BASE_DIR / 'data' / 'uploads'
        uploads_dir.mkdir(parents=True, exist_ok=True)

        document_id = f"doc-{uuid.uuid4().hex[:12]}"
        file_path = uploads_dir / f"{document_id}.pdf"
        uploaded.save(str(file_path))

        logger.info(f"Uploaded document: {uploaded.filename} -> {document_id}")
```

```
1149:
1150:         return jsonify({
1151:             'status': 'success',
1152:             'document_id': document_id,
1153:             'filename': uploaded.filename,
1154:             'path': str(file_path),
1155:             'timestamp': datetime.now().isoformat(),
1156:         })
1157:
1158:     except Exception as exc:
1159:         logger.error(f"Failed to upload document: {exc}")
1160:         return jsonify({'error': str(exc)}), 500
1161:
1162: @app.route('/api/admin/run-analysis', methods=['POST'])
1163: @require_auth
1164: def trigger_pipeline_analysis():
1165:     """
1166:     Trigger pipeline analysis for uploaded document.
1167:
1168:     Request body:
1169:         {
1170:             "document_id": "doc-abc123",
1171:             "municipality": "Optional municipality name",
1172:             "settings": {
1173:                 "phase_timeout": 300,
1174:                 "enable_cache": true,
1175:                 "enable_parallel": true,
1176:                 "log_level": "INFO"
1177:             }
1178:         }
1179:
1180:     Returns job_id for WebSocket progress tracking.
1181:     """
1182:     try:
1183:         payload = request.get_json()
1184:         document_id = payload.get('document_id')
1185:
1186:         if not document_id:
1187:             return jsonify({'error': 'Missing document_id'}), 400
1188:
1189:         uploads_dir = BASE_DIR / 'data' / 'uploads'
1190:         pdf_path = uploads_dir / f"{document_id}.pdf"
1191:
1192:         if not pdf_path.exists():
1193:             return jsonify({'error': f'Document not found: {document_id}'}), 404
1194:
1195:         municipality = payload.get('municipality', document_id)
1196:         settings = payload.get('settings', {})
1197:
1198:         job_id = data_service.start_pipeline_job(
1199:             pdf_path=pdf_path,
1200:             municipality=municipality,
1201:             settings=settings,
1202:         )
1203:
1204:         logger.info(f"Pipeline analysis started: job_id={job_id}, doc={document_id}")
```

```
1205:
1206:         return jsonify({
1207:             'status': 'success',
1208:             'job_id': job_id,
1209:             'document_id': document_id,
1210:             'municipality': municipality,
1211:             'timestamp': datetime.now().isoformat(),
1212:         })
1213:
1214:     except Exception as exc:
1215:         logger.error(f"Failed to trigger pipeline analysis: {exc}")
1216:         return jsonify({'error': str(exc)}), 500
1217:
1218: # ============================================================================
1219: # WEBSOCKET HANDLERS FOR REAL-TIME UPDATES
1220: # ============================================================================
1221:
1222: @socketio.on('connect')
1223: def handle_connect() -> None:
1224:     """Handle WebSocket connection"""
1225:     logger.info(f"Client connected: {request.sid}")
1226:     emit('connection_response', {'status': 'connected'})
1227:
1228: @socketio.on('disconnect')
1229: def handle_disconnect() -> None:
1230:     """Handle WebSocket disconnection"""
1231:     logger.info(f"Client disconnected: {request.sid}")
1232:
1233: @socketio.on('subscribe_region')
1234: def handle_subscribe_region(data) -> None:
1235:     """Subscribe to region updates"""
1236:     region_id = data.get('region_id')
1237:     logger.info(f"Client {request.sid} subscribed to region: {region_id}")
1238:
1239:     # Send initial data
1240:     region = data_service.get_region_detail(region_id)
1241:     emit('region_update', region)
1242:
1243: # ============================================================================
1244: # ERROR HANDLERS
1245: # ============================================================================
1246:
1247: @app.errorhandler(HTTPException)
1248: def handle_http_exception(e):
1249:     """Handle HTTP exceptions"""
1250:     return jsonify({
1251:         'error': e.description,
1252:         'status_code': e.code
1253:     }), e.code
1254:
1255: @app.errorhandler(Exception)
1256: def handle_exception(e):
1257:     """Handle general exceptions"""
1258:     logger.error(f"Unhandled exception: {e}")
1259:     return jsonify({
1260:         'error': 'Internal server error',
```

```
        'message': str(e)
    }), 500

# ============================================================================
# RECOMMENDATION ENDPOINTS
# ============================================================================

@app.route('/api/v1/recommendations/micro', methods=['POST'])
@rate_limit
def generate_micro_recommendations():
    """
    Generate MICRO-level recommendations

    Request Body:
        {
            "scores": {
                "PA01-DIM01": 1.2,
                "PA02-DIM02": 1.5,
                ...
            },
            "context": {}  // Optional
        }

    Returns:
        RecommendationSet with MICRO recommendations
    """
    if not recommendation_engine:
        return jsonify({'error': 'Recommendation engine not available'}), 503

    try:
        data = request.get_json()
        scores = data.get('scores', {})
        context = data.get('context', {})

        if not scores:
            return jsonify({'error': 'Missing scores'}), 400

        rec_set = recommendation_engine.generate_micro_recommendations(scores, context)

        return jsonify({
            'status': 'success',
            'data': rec_set.to_dict(),
            'timestamp': datetime.now().isoformat()
        })

    except Exception as e:
        logger.error(f"Failed to generate MICRO recommendations: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/api/v1/recommendations/meso', methods=['POST'])
@rate_limit
def generate_meso_recommendations():
    """
    Generate MESO-level recommendations

    Request Body:
```

```
        {
            "cluster_data": {
                "CL01": {"score": 72.0, "variance": ParameterLoaderV2.get("farfan_core.api.api_server.DataService.get_evidence_stream", "auto_param_L865_52"
, 0.25), "weak_pa": "PA02"},
                ...
            },
            "context": {}  // Optional
        }

    Returns:
        RecommendationSet with MESO recommendations
    """
    if not recommendation_engine:
        return jsonify({'error': 'Recommendation engine not available'}), 503

    try:
        data = request.get_json()
        cluster_data = data.get('cluster_data', {})
        context = data.get('context', {})

        if not cluster_data:
            return jsonify({'error': 'Missing cluster_data'}), 400

        rec_set = recommendation_engine.generate_meso_recommendations(cluster_data, context)

        return jsonify({
            'status': 'success',
            'data': rec_set.to_dict(),
            'timestamp': datetime.now().isoformat()
        })

    except Exception as e:
        logger.error(f"Failed to generate MESO recommendations: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/api/v1/recommendations/macro', methods=['POST'])
@rate_limit
def generate_macro_recommendations():
    """
    Generate MACRO-level recommendations

    Request Body:
        {
            "macro_data": {
                "macro_band": "SATISFACTORIO",
                "clusters_below_target": ["CL02", "CL03"],
                "variance_alert": "MODERADA",
                "priority_micro_gaps": ["PA01-DIM05", "PA04-DIM04"]
            },
            "context": {}  // Optional
        }

    Returns:
        RecommendationSet with MACRO recommendations
    """
    if not recommendation_engine:
```

```
        return jsonify({'error': 'Recommendation engine not available'}), 503

    try:
        data = request.get_json()
        macro_data = data.get('macro_data', {})
        context = data.get('context', {})

        if not macro_data:
            return jsonify({'error': 'Missing macro_data'}), 400

        rec_set = recommendation_engine.generate_macro_recommendations(macro_data, context)

        return jsonify({
            'status': 'success',
            'data': rec_set.to_dict(),
            'timestamp': datetime.now().isoformat()
        })

    except Exception as e:
        logger.error(f"Failed to generate MACRO recommendations: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/api/v1/recommendations/all', methods=['POST'])
@rate_limit
def generate_all_recommendations():
    """
    Generate recommendations at all levels (MICRO, MESO, MACRO)

    Request Body:
        {
            "micro_scores": {...},
            "cluster_data": {...},
            "macro_data": {...},
            "context": {}  // Optional
        }

    Returns:
        Dictionary with MICRO, MESO, and MACRO recommendation sets
    """
    if not recommendation_engine:
        return jsonify({'error': 'Recommendation engine not available'}), 503

    try:
        data = request.get_json()
        micro_scores = data.get('micro_scores', {})
        cluster_data = data.get('cluster_data', {})
        macro_data = data.get('macro_data', {})
        context = data.get('context', {})

        all_recs = recommendation_engine.generate_all_recommendations(
            micro_scores, cluster_data, macro_data, context
        )

        return jsonify({
            'status': 'success',
            'data': {
```

```
                'MICRO': all_recs['MICRO'].to_dict(),
                'MESO': all_recs['MESO'].to_dict(),
                'MACRO': all_recs['MACRO'].to_dict()
            },
            'summary': {
                'MICRO': {
                    'total_rules': all_recs['MICRO'].total_rules_evaluated,
                    'matched': all_recs['MICRO'].rules_matched
                },
                'MESO': {
                    'total_rules': all_recs['MESO'].total_rules_evaluated,
                    'matched': all_recs['MESO'].rules_matched
                },
                'MACRO': {
                    'total_rules': all_recs['MACRO'].total_rules_evaluated,
                    'matched': all_recs['MACRO'].rules_matched
                }
            },
            'timestamp': datetime.now().isoformat()
        })

    except Exception as e:
        logger.error(f"Failed to generate all recommendations: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/api/v1/recommendations/rules/info', methods=['GET'])
@rate_limit
@cache_response(timeout=600)
def get_rules_info():
    """
    Get information about loaded recommendation rules

    Returns:
        Statistics about loaded rules
    """
    if not recommendation_engine:
        return jsonify({'error': 'Recommendation engine not available'}), 503

    try:
        return jsonify({
            'status': 'success',
            'data': {
                'version': recommendation_engine.rules.get('version'),
                'total_rules': len(recommendation_engine.rules.get('rules', [])),
                'by_level': {
                    'MICRO': len(recommendation_engine.rules_by_level['MICRO']),
                    'MESO': len(recommendation_engine.rules_by_level['MESO']),
                    'MACRO': len(recommendation_engine.rules_by_level['MACRO'])
                },
                'rules_path': str(recommendation_engine.rules_path),
                'schema_path': str(recommendation_engine.schema_path)
            },
            'timestamp': datetime.now().isoformat()
        })

    except Exception as e:
```

```
        logger.error(f"Failed to get rules info: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/api/v1/recommendations/reload', methods=['POST'])
@require_auth
def reload_rules():
    """
    Reload recommendation rules from disk (admin only)

    Returns:
        Success status
    """
    if not recommendation_engine:
        return jsonify({'error': 'Recommendation engine not available'}), 503

    try:
        recommendation_engine.reload_rules()

        return jsonify({
            'status': 'success',
            'message': 'Rules reloaded successfully',
            'total_rules': len(recommendation_engine.rules.get('rules', [])),
            'timestamp': datetime.now().isoformat()
        })

    except Exception as e:
        logger.error(f"Failed to reload rules: {e}")
        return jsonify({'error': str(e)}), 500

# ============================================================================
# MAIN
# ============================================================================

def main() -> None:
    """Run API server"""
    logger.info("=" * 80)
    logger.info("AtroZ Dashboard API Server")
    logger.info("=" * 80)
    logger.info(f"CORS Origins: {APIConfig.CORS_ORIGINS}")
    logger.info(f"Rate Limiting: {APIConfig.RATE_LIMIT_ENABLED}")
    logger.info(f"Caching: {APIConfig.CACHE_ENABLED}")
    logger.info("=" * 80)

    # Run server
    socketio.run(
        app,
        host='0.0.0.0',
        port=int(os.getenv('ATROZ_API_PORT', '5000')),
        debug=os.getenv('ATROZ_DEBUG', 'false').lower() == 'true'
    )

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

```
1540: ================================================================================
1541: FILE: src/farfan_pipeline/api/auth_admin.py
1542: ================================================================================
1543:
1544: """
1545: AtroZ Admin Authentication Module
1546: Minimal but secure authentication for admin panel access
1547: """
1548:
1549: import hashlib
1550: import logging
1551: import secrets
1552: import time
1553: from dataclasses import dataclass
1554: from datetime import datetime, timedelta
1555: from farfan_pipeline.core.calibration.decorators import calibrated_method
1556:
1557: logger = logging.getLogger(__name__)
1558:
1559:
1560: @dataclass
1561: class AdminSession:
1562:     """Represents an active admin session"""
1563:     session_id: str
1564:     username: str
1565:     created_at: datetime
1566:     last_activity: datetime
1567:     ip_address: str
1568:
1569:     @calibrated_method("farfan_core.api.auth_admin.AdminSession.is_expired")
1570:     def is_expired(self, timeout_minutes: int = 60) -> bool:
1571:         """Check if session has expired"""
1572:         return datetime.now() - self.last_activity > timedelta(minutes=timeout_minutes)
1573:
1574:     @calibrated_method("farfan_core.api.auth_admin.AdminSession.update_activity")
1575:     def update_activity(self) -> None:
1576:         """Update last activity timestamp"""
1577:         self.last_activity = datetime.now()
1578:
1579:
1580: class AdminAuthenticator:
1581:     """
1582:     Simple but secure authentication system for admin panel.
1583:
1584:     Security features:
1585:     - Password hashing with salt
1586:     - Session management with timeout
1587:     - Rate limiting on login attempts
1588:     - IP-based session tracking
1589:     """
1590:
1591:     def __init__(self, session_timeout_minutes: int = 60) -> None:
1592:         self.session_timeout = session_timeout_minutes
1593:         self.sessions: dict[str, AdminSession] = {}
1594:         self.login_attempts: dict[str, list] = {}
1595:
```

```
        # Default credentials (should be changed in production)
        # Default password: "atroz_admin_2024"
        self.users = {
            "admin": {
                "password_hash": self._hash_password("atroz_admin_2024", "default_salt"),
                "salt": "default_salt",
                "role": "administrator"
            }
        }

        logger.info("Admin authenticator initialized")

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

    @calibrated_method("farfan_core.api.auth_admin.AdminAuthenticator._generate_session_id")
    def _generate_session_id(self) -> str:
        """Generate secure random session ID"""
        return secrets.token_urlsafe(32)

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

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

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

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

        return True

    @calibrated_method("farfan_core.api.auth_admin.AdminAuthenticator.authenticate")
    def authenticate(self, username: str, password: str, ip_address: str) -> str | None:
        """
        Authenticate user and create session.

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

        Returns:
            Session ID if authentication successful, None otherwise
```

```
        """
        # Check rate limit
        if not self._check_rate_limit(ip_address):
            return None

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

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

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

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

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

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

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

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

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

        session = self.sessions[session_id]

        # Check if expired
        if session.is_expired(self.session_timeout):
```

```
            logger.info(f"Session expired for user: {session.username}")
            del self.sessions[session_id]
            return False

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

        # Update activity
        session.update_activity()
        return True

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

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

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

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

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

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

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

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

```

```
    @calibrated_method("farfan_core.api.auth_admin.AdminAuthenticator.change_password")
    def change_password(self, username: str, old_password: str, new_password: str) -> bool:
        """Change user password"""
        if username not in self.users:
            return False

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

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

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

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

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


# Global authenticator instance
_authenticator: AdminAuthenticator | None = None


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


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

    from flask import jsonify, request

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

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

        auth = get_authenticator()
        ip_address = request.remote_addr

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

```
1820:
1821:         return func(*args, **kwargs)
1822:
1823:     return wrapper
1824:
1825:
1826:
1827: ================================================================================
1828: FILE: src/farfan_pipeline/api/dashboard_data_service.py
1829: ================================================================================
1830:
1831: from __future__ import annotations
1832:
1833: import json
1834: import logging
1835: from datetime import datetime
1836: from pathlib import Path
1837: from typing import Any, Iterable, Sequence
1838:
1839: logger = logging.getLogger(__name__)
1840:
1841:
1842: class DashboardDataService:
1843:     """Transforms pipeline artifacts into dashboard-friendly payloads."""
1844:
1845:     def __init__(self, jobs_dir: Path) -> None:
1846:         self.jobs_dir = jobs_dir
1847:
1848:     # ------------------------------------------------------------------
1849:     # Public interface
1850:     # ------------------------------------------------------------------
1851:
1852:     def summarize_region(
1853:         self,
1854:         record: dict[str, Any],
1855:     ) -> tuple[dict[str, Any], dict[str, Any]]:
1856:         """Build lightweight region summary and context for downstream use."""
1857:         report = self._load_report(record)
1858:         macro_detail = self._extract_macro_detail(report, record)
1859:         clusters = self._extract_clusters(report, record)
1860:         scores = self._build_scores(record, macro_detail)
1861:         indicators = self._build_indicators(record, macro_detail, scores, clusters)
1862:         metadata = self._build_metadata(record, macro_detail)
1863:
1864:         summary = {
1865:             'id': record.get('id'),
1866:             'job_id': record.get('job_id'),
1867:             'name': (record.get('name') or '').upper(),
1868:             'municipality': record.get('municipality'),
1869:             'coordinates': self._build_coordinates(record),
1870:             'metadata': metadata,
1871:             'scores': scores,
1872:             'clusterScores': {entry['cluster_id']: entry['score_percent'] for entry in clusters if entry.get('cluster_id')},
1873:             'connections': list(record.get('connections') or []),
1874:             'indicators': indicators,
1875:             'macroBand': metadata.get('macroBand'),
```

```
1876:             'updated_at': record.get('updated_at'),
1877:         }
1878:
1879:         context = {
1880:             'report': report,
1881:             'macro': macro_detail,
1882:             'clusters': clusters,
1883:         }
1884:         return summary, context
1885:
1886:     def build_region_detail(
1887:         self,
1888:         record: dict[str, Any],
1889:         summary: dict[str, Any],
1890:         context: dict[str, Any],
1891:         region_evidence: Iterable[dict[str, Any]] | None = None,
1892:     ) -> dict[str, Any]:
1893:         """Build full region payload with macro/meso/micro breakdown."""
1894:         report = context.get('report') or {}
1895:         macro_detail = context.get('macro') or self._extract_macro_detail(report, record)
1896:         clusters = context.get('clusters') or self._extract_clusters(report, record)
1897:         question_matrix = self._extract_question_matrix(report)
1898:         recommendations = self._extract_recommendations(report, macro_detail, clusters)
1899:         evidence_stream = self._merge_evidence(region_evidence, question_matrix, record)
1900:
1901:         detailed = dict(summary)
1902:         detailed['macro'] = macro_detail
1903:         detailed['meso'] = clusters
1904:         detailed['micro'] = question_matrix
1905:         detailed['detailed_analysis'] = {
1906:             'cluster_breakdown': self._to_cluster_breakdown(clusters),
1907:             'question_matrix': question_matrix,
1908:             'recommendations': recommendations,
1909:             'evidence': evidence_stream,
1910:         }
1911:         return detailed
1912:
1913:     def extract_question_matrix(self, report: dict[str, Any]) -> list[dict[str, Any]]:
1914:         """Expose normalized question matrix for other services."""
1915:         return self._extract_question_matrix(report)
1916:
1917:     def normalize_evidence_stream(
1918:         self,
1919:         evidence: Iterable[dict[str, Any]],
1920:         limit: int = 50,
1921:     ) -> list[dict[str, Any]]:
1922:         """Normalize persisted evidence items for ticker display."""
1923:         normalized = [self._normalize_evidence_item(item) for item in evidence if item]
1924:         normalized = [item for item in normalized if item]
1925:         normalized.sort(key=lambda item: item.get('timestamp') or '', reverse=True)
1926:         return normalized[:limit]
1927:
1928:     # ------------------------------------------------------------------
1929:     # Report parsing helpers
1930:     # ------------------------------------------------------------------
1931:
```

```
1932:     def _load_report(self, record: dict[str, Any]) -> dict[str, Any]:
1933:         """Load latest pipeline report for region if available."""
1934:         job_id = record.get('latest_report') or record.get('job_id')
1935:         candidate_paths: list[Path] = []
1936:
1937:         report_path = record.get('report_path')
1938:         if report_path:
1939:             candidate_paths.append(Path(report_path))
1940:         if job_id:
1941:             candidate_paths.append(self.jobs_dir / f"{job_id}.json")
1942:
1943:         for path in candidate_paths:
1944:             if not path or not path.exists():
1945:                 continue
1946:             try:
1947:                 with open(path, 'r', encoding='utf-8') as handle:
1948:                     payload = json.load(handle)
1949:                 if isinstance(payload, dict):
1950:                     report = payload.get('report') if 'report' in payload else payload
1951:                     if isinstance(report, dict):
1952:                         return report
1953:             except Exception as exc:  # pragma: no cover - best effort
1954:                 logger.warning("Failed to load dashboard report %s: %s", path, exc)
1955:         return {}
1956:
1957:     def _build_coordinates(self, record: dict[str, Any]) -> dict[str, float]:
1958:         coords = record.get('coordinates') or {}
1959:         x = self._first_number([coords.get('x'), coords.get('lng'), coords.get('lon')], default=50.0)
1960:         y = self._first_number([coords.get('y'), coords.get('lat')], default=50.0)
1961:         return {'x': float(x), 'y': float(y)}
1962:
1963:     def _build_metadata(self, record: dict[str, Any], macro_detail: dict[str, Any]) -> dict[str, Any]:
1964:         stats = dict(record.get('stats') or {})
1965:         if 'departments' in stats and not isinstance(stats['departments'], list):
1966:             stats['departments'] = list(self._ensure_list(stats['departments']))
1967:         if macro_detail.get('band'):
1968:             stats['macroBand'] = macro_detail['band']
1969:         return stats
1970:
1971:     def _build_scores(self, record: dict[str, Any], macro_detail: dict[str, Any]) -> dict[str, Any]:
1972:         scores = dict(record.get('scores') or {})
1973:         if macro_detail.get('score_percent') is not None:
1974:             scores.setdefault('overall', macro_detail['score_percent'])
1975:         scores.setdefault('lastUpdated', record.get('updated_at'))
1976:         return scores
1977:
1978:     def _build_indicators(
1979:         self,
1980:         record: dict[str, Any],
1981:         macro_detail: dict[str, Any],
1982:         scores: dict[str, Any],
1983:         clusters: Sequence[dict[str, Any]],
1984:     ) -> dict[str, Any]:
1985:         if record.get('indicators'):
1986:             return dict(record['indicators'])
1987:
```

```
        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
```

```
2212:
2213:     def _normalize_question_entry(self, item: dict[str, Any]) -> dict[str, Any] | None:
2214:         question_id = item.get('question_id') or item.get('id') or item.get('code')
2215:         if not question_id:
2216:             return None
2217:         text = item.get('text')
2218:         if not text and isinstance(item.get('metadata'), dict):
2219:             text = item['metadata'].get('title') or item['metadata'].get('question_text')
2220:         text = text or f'Pregunta {question_id}'
2221:
2222:         metadata = item.get('metadata') if isinstance(item.get('metadata'), dict) else {}
2223:         policy_area = metadata.get('policy_area_id') or metadata.get('policy_area')
2224:         dimension = metadata.get('dimension_id') or metadata.get('dimension')
2225:         if not policy_area or not dimension:
2226:             inferred_area, inferred_dimension = self._parse_question_identifier(str(question_id))
2227:             policy_area = policy_area or inferred_area
2228:             dimension = dimension or inferred_dimension
2229:
2230:         score = self._first_number([
2231:             item.get('score'),
2232:             item.get('adjusted_score'),
2233:             item.get('value'),
2234:         ])
2235:         score_percent = None
2236:         if score is not None:
2237:             if score <= 1.0:
2238:                 score_percent = self._to_percent(score)
2239:             elif score <= 3.0:
2240:                 score_percent = self._sanitize_percent((score / 3.0) * 100.0)
2241:             else:
2242:                 score_percent = self._sanitize_percent(score)
2243:
2244:         evidence = self._normalize_evidence_list(item.get('evidence'), question_id)
2245:         recommendations = self._normalize_recommendations(item.get('recommendations') or item.get('recommendation'))
2246:
2247:         return {
2248:             'id': str(question_id),
2249:             'text': text,
2250:             'score': score,
2251:             'score_percent': score_percent,
2252:             'category': policy_area,
2253:             'dimension': dimension,
2254:             'evidence': evidence,
2255:             'recommendations': recommendations,
2256:         }
2257:
2258:     def _parse_question_identifier(self, identifier: str) -> tuple[str | None, str | None]:
2259:         cleaned = identifier.replace('_', '-').upper()
2260:         parts = cleaned.split('-')
2261:         policy_area = None
2262:         dimension = None
2263:         for part in parts:
2264:             if part.startswith('PA') and part[2:].isdigit():
2265:                 policy_area = part
2266:             elif part.startswith('DIM') and part[3:].isdigit():
2267:                 dimension = part
```

```
2268:             elif part.startswith('D') and part[1:].isdigit() and not dimension:
2269:                 dimension = f'DIM{part[1:]}'
2270:         return policy_area, dimension
2271:
2272:     def _normalize_recommendations(self, value: Any) -> list[str]:
2273:         if value is None:
2274:             return []
2275:         if isinstance(value, list):
2276:             return [str(item) for item in value]
2277:         return [str(value)]
2278:
2279:     def _extract_recommendations(
2280:         self,
2281:         report: dict[str, Any],
2282:         macro_detail: dict[str, Any],
2283:         clusters: Sequence[dict[str, Any]],
2284:     ) -> list[dict[str, Any]]:
2285:         raw_recs = report.get('recommendations') if isinstance(report, dict) else None
2286:         if isinstance(raw_recs, dict):
2287:             items = raw_recs.get('items')
2288:             raw_recs = items if isinstance(items, list) else list(raw_recs.values())
2289:         recommendations: list[dict[str, Any]] = []
2290:         if isinstance(raw_recs, list):
2291:             for entry in raw_recs:
2292:                 normalized = self._normalize_recommendation_entry(entry)
2293:                 if normalized:
2294:                     recommendations.append(normalized)
2295:         elif raw_recs:
2296:             normalized = self._normalize_recommendation_entry(raw_recs)
2297:             if normalized:
2298:                 recommendations.append(normalized)
2299:
2300:         if not recommendations:
2301:             for gap in macro_detail.get('systemic_gaps') or []:
2302:                 recommendations.append(
2303:                     {
2304:                         'priority': 'ALTA',
2305:                         'text': str(gap),
2306:                         'category': 'MACRO',
2307:                         'impact': 'HIGH',
2308:                     }
2309:                 )
2310:         if not recommendations and clusters:
2311:             worst = min(
2312:                 (cluster for cluster in clusters if cluster.get('score_percent') is not None),
2313:                 key=lambda cluster: cluster['score_percent'],
2314:                 default=None,
2315:             )
2316:             if worst:
2317:                 recommendations.append(
2318:                     {
2319:                         'priority': 'MEDIA',
2320:                         'text': f'Reforzar intervenciones en {worst["name"]}',
2321:                         'category': 'MESO',
2322:                         'impact': 'MEDIUM',
2323:                     }
```

```
                )
        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]



================================================================================
FILE: src/farfan_pipeline/api/dashboard_server.py
================================================================================

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 farfan_pipeline.core.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 MarÃ-a", "score": 91, "x": 35, "y": 10, "municipalities": 15},
    {"id": "pacifico_medio", "name": "PacÃ-fico 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": "UrabÃ¡ Antioqueã±o", "score": 69, "x": 20, "y": 25, "municipalities": 8},
    {"id": "choco", "name": "ChocÃ³", "score": 55, "x": 8, "y": 30, "municipalities": 12},
    {"id": "macarena", "name": "Macarena - Guaviare", "score": 82, "x": 45, "y": 50, "municipalities": 12},
    {"id": "pacifico_nariÃ±ense", "name": "PacÃ-fico NariÃ±ense", "score": 60, "x": 5, "y": 70, "municipalities": 11},
    {"id": "cuenca_caguan", "name": "Cuenca del CaguÃ¡n", "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 BolÃ-var", "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 CÃ³rdoba", "score": 63, "x": 22, "y": 18, "municipalities": 5},
    {"id": "alto_patia", "name": "Alto PatÃ-a", "score": 80, "x": 15, "y": 65, "municipalities": 24}
]

```

```
# Evidence stream - will be populated by pipeline analysis
EVIDENCE_STREAM = [
    {"source": "PDT SecciÃ³n 3.2", "page": 45, "text": "ImplementaciÃ³n de estrategias municipales", "region": "arauca"},
    {"source": "PDT CapÃtulo 4", "page": 67, "text": "ArticulaciÃ³n con DecÃ¡logo DDHH", "region": "catatumbo"},
    {"source": "Anexo TÃ©cnico", "page": 112, "text": "Indicadores de cumplimiento territorial", "region": "montes_maria"},
    {"source": "PDT SecciÃ³n 5.1", "page": 89, "text": "ProyecciÃ³n territorial 2030", "region": "putumayo"},
    {"source": "PATR CapÃtulo 2", "page": 34, "text": "Cadenas de valor agropecuarias", "region": "bajo_cauca"},
    {"source": "PDT SecciÃ³n 6.3", "page": 156, "text": "Mecanismos de participaciÃ³n 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')
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)


================================================================================
FILE: src/farfan_pipeline/api/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
from farfan_pipeline.core.calibration.decorators import calibrated_method


class PDETSubregion(Enum):
    """16 PDET Subregions"""
    ALTO_PATIA = "Alto PatÃ-a y Norte del Cauca"
    ARAUCA = "Arauca"
    BAJO_CAUCA = "Bajo Cauca y Nordeste AntioqueÃ±o"
    CAGUAN = "Cuenca del CaguÃ¡n y Piedemonte CaqueteÃ±o"
    CATATUMBO = "Catatumbo"
    CHOCO = "ChocÃ³"
    MACARENA = "Macarena-Guaviare"
    MONTES_MARIA = "Montes de MarÃ-a"
    PACIFICO_MEDIO = "PacÃ-fico Medio"
    PACIFICO_NARINENSE = "PacÃ-fico y Frontera NariÃ±ense"
    PUTUMAYO = "Putumayo"
    SIERRA_NEVADA = "Sierra Nevada - PerijÃ¡ - Zona Bananera"
    SUR_BOLIVAR = "Sur de BolÃ-var"
    SUR_CORDOBA = "Sur de CÃ³rdoba"
    SUR_TOLIMA = "Sur del Tolima"
    URABA = "UrabÃ¡ AntioqueÃ±o"


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

```

```
2772:
2773: # Complete PDET Municipality Dataset (170 municipalities)
2774: PDET_MUNICIPALITIES: list[PDETMunicipality] = [
2775:     # ALTO PATÃ\215A Y NORTE DEL CAUCA (24 municipalities)
2776:     PDETMunicipality("Argelia", "Cauca", PDETSubregion.ALTO_PATIA, 31000, 661.0, "19050"),
2777:     PDETMunicipality("Balboa", "Cauca", PDETSubregion.ALTO_PATIA, 22000, 388.0, "19075"),
2778:     PDETMunicipality("Buenos Aires", "Cauca", PDETSubregion.ALTO_PATIA, 32000, 519.0, "19100"),
2779:     PDETMunicipality("CajibÃ-o", "Cauca", PDETSubregion.ALTO_PATIA, 38000, 440.0, "19110"),
2780:     PDETMunicipality("Caldono", "Cauca", PDETSubregion.ALTO_PATIA, 31000, 249.0, "19137"),
2781:     PDETMunicipality("Caloto", "Cauca", PDETSubregion.ALTO_PATIA, 40000, 350.0, "19142"),
2782:     PDETMunicipality("Corinto", "Cauca", PDETSubregion.ALTO_PATIA, 33000, 273.0, "19212"),
2783:     PDETMunicipality("El Tambo", "Cauca", PDETSubregion.ALTO_PATIA, 50000, 3213.0, "19256"),
2784:     PDETMunicipality("JambalÃ³", "Cauca", PDETSubregion.ALTO_PATIA, 17000, 51.0, "19364"),
2785:     PDETMunicipality("Mercaderes", "Cauca", PDETSubregion.ALTO_PATIA, 21000, 604.0, "19418"),
2786:     PDETMunicipality("Miranda", "Cauca", PDETSubregion.ALTO_PATIA, 42000, 597.0, "19455"),
2787:     PDETMunicipality("Morales", "Cauca", PDETSubregion.ALTO_PATIA, 28000, 580.0, "19473"),
2788:     PDETMunicipality("PatÃ-a", "Cauca", PDETSubregion.ALTO_PATIA, 37000, 834.0, "19513"),
2789:     PDETMunicipality("PiendamÃ³", "Cauca", PDETSubregion.ALTO_PATIA, 44000, 116.0, "19532"),
2790:     PDETMunicipality("Santander de Quilichao", "Cauca", PDETSubregion.ALTO_PATIA, 95000, 543.0, "19693"),
2791:     PDETMunicipality("SuÃ¡rez", "Cauca", PDETSubregion.ALTO_PATIA, 20000, 364.0, "19698"),
2792:     PDETMunicipality("ToribÃ-o", "Cauca", PDETSubregion.ALTO_PATIA, 31000, 186.0, "19821"),
2793:     PDETMunicipality("Cumbitara", "NariÃ±o", PDETSubregion.ALTO_PATIA, 16000, 600.0, "52227"),
2794:     PDETMunicipality("El Rosario", "NariÃ±o", PDETSubregion.ALTO_PATIA, 12000, 558.0, "52258"),
2795:     PDETMunicipality("Leiva", "NariÃ±o", PDETSubregion.ALTO_PATIA, 13000, 395.0, "52381"),
2796:     PDETMunicipality("Los Andes", "NariÃ±o", PDETSubregion.ALTO_PATIA, 15000, 434.0, "52427"),
2797:     PDETMunicipality("Policarpa", "NariÃ±o", PDETSubregion.ALTO_PATIA, 17000, 624.0, "52585"),
2798:     PDETMunicipality("Florida", "Valle del Cauca", PDETSubregion.ALTO_PATIA, 58000, 517.0, "76275"),
2799:     PDETMunicipality("Pradera", "Valle del Cauca", PDETSubregion.ALTO_PATIA, 61000, 273.0, "76563"),
2800:
2801:     # ARAUCA (4 municipalities)
2802:     PDETMunicipality("Arauquita", "Arauca", PDETSubregion.ARAUCA, 45000, 3828.0, "81065"),
2803:     PDETMunicipality("Fortul", "Arauca", PDETSubregion.ARAUCA, 27000, 1997.0, "81300"),
2804:     PDETMunicipality("Saravena", "Arauca", PDETSubregion.ARAUCA, 53000, 1879.0, "81736"),
2805:     PDETMunicipality("Tame", "Arauca", PDETSubregion.ARAUCA, 53000, 5278.0, "81794"),
2806:
2807:     # BAJO CAUCA Y NORDESTE ANTIOQUEÃ\221O (13 municipalities)
2808:     PDETMunicipality("CÃ¡ceres", "Antioquia", PDETSubregion.BAJO_CAUCA, 39000, 2273.0, "05120"),
2809:     PDETMunicipality("Caucasia", "Antioquia", PDETSubregion.BAJO_CAUCA, 104000, 1842.0, "05154"),
2810:     PDETMunicipality("El Bagre", "Antioquia", PDETSubregion.BAJO_CAUCA, 53000, 1824.0, "05250"),
2811:     PDETMunicipality("NechÃ-", "Antioquia", PDETSubregion.BAJO_CAUCA, 29000, 2803.0, "05495"),
2812:     PDETMunicipality("TarazÃ¡", "Antioquia", PDETSubregion.BAJO_CAUCA, 45000, 1923.0, "05790"),
2813:     PDETMunicipality("Zaragoza", "Antioquia", PDETSubregion.BAJO_CAUCA, 30000, 900.0, "05895"),
2814:     PDETMunicipality("Amalfi", "Antioquia", PDETSubregion.BAJO_CAUCA, 23000, 1224.0, "05030"),
2815:     PDETMunicipality("AnorÃ-", "Antioquia", PDETSubregion.BAJO_CAUCA, 18000, 1445.0, "05040"),
2816:     PDETMunicipality("Remedios", "Antioquia", PDETSubregion.BAJO_CAUCA, 29000, 1985.0, "05604"),
2817:     PDETMunicipality("Segovia", "Antioquia", PDETSubregion.BAJO_CAUCA, 40000, 1234.0, "05756"),
2818:     PDETMunicipality("Valdivia", "Antioquia", PDETSubregion.BAJO_CAUCA, 20000, 1088.0, "05854"),
2819:     PDETMunicipality("VegachÃ-", "Antioquia", PDETSubregion.BAJO_CAUCA, 9000, 582.0, "05858"),
2820:     PDETMunicipality("YondÃ³", "Antioquia", PDETSubregion.BAJO_CAUCA, 18000, 1635.0, "05893"),
2821:
2822:     # CUENCA DEL CAGUÃ\201N Y PIEDEMONTE CAQUETEÃ\221O (17 municipalities)
2823:     PDETMunicipality("Albania", "CaquetÃ¡", PDETSubregion.CAGUAN, 5000, 1149.0, "18029"),
2824:     PDETMunicipality("BelÃ©n de los AndaquÃ-es", "CaquetÃ¡", PDETSubregion.CAGUAN, 11000, 1168.0, "18094"),
2825:     PDETMunicipality("Cartagena del ChairÃ¡", "CaquetÃ¡", PDETSubregion.CAGUAN, 35000, 12704.0, "18150"),
2826:     PDETMunicipality("Curillo", "CaquetÃ¡", PDETSubregion.CAGUAN, 11000, 1463.0, "18205"),
2827:     PDETMunicipality("El Doncello", "CaquetÃ¡", PDETSubregion.CAGUAN, 25000, 1195.0, "18247"),
```

```
2828:     PDETMunicipality("El Paujil", "CaquetÃ¡", PDETSubregion.CAGUAN, 21000, 907.0, "18256"),
2829:     PDETMunicipality("Florencia", "CaquetÃ¡", PDETSubregion.CAGUAN, 180000, 2292.0, "18001"),
2830:     PDETMunicipality("La MontaÃ±ita", "CaquetÃ¡", PDETSubregion.CAGUAN, 24000, 1462.0, "18410"),
2831:     PDETMunicipality("MilÃ¡n", "CaquetÃ¡", PDETSubregion.CAGUAN, 11000, 940.0, "18460"),
2832:     PDETMunicipality("Morelia", "CaquetÃ¡", PDETSubregion.CAGUAN, 4000, 1386.0, "18479"),
2833:     PDETMunicipality("Puerto Rico", "CaquetÃ¡", PDETSubregion.CAGUAN, 36000, 15224.0, "18592"),
2834:     PDETMunicipality("San JosÃ© del Fragua", "CaquetÃ¡", PDETSubregion.CAGUAN, 14000, 3938.0, "18610"),
2835:     PDETMunicipality("San Vicente del CaguÃ¡n", "CaquetÃ¡", PDETSubregion.CAGUAN, 64000, 24466.0, "18753"),
2836:     PDETMunicipality("Solano", "CaquetÃ¡", PDETSubregion.CAGUAN, 22000, 42625.0, "18756"),
2837:     PDETMunicipality("Solita", "CaquetÃ¡", PDETSubregion.CAGUAN, 14000, 9057.0, "18785"),
2838:     PDETMunicipality("ValparaÃ-so", "CaquetÃ¡", PDETSubregion.CAGUAN, 16000, 1231.0, "18860"),
2839:     PDETMunicipality("Algeciras", "Huila", PDETSubregion.CAGUAN, 23000, 626.0, "41026"),
2840:
2841:     # CATATUMBO (8 municipalities)
2842:     PDETMunicipality("ConvenciÃ³n", "Norte de Santander", PDETSubregion.CATATUMBO, 19000, 1171.0, "54206"),
2843:     PDETMunicipality("El Carmen", "Norte de Santander", PDETSubregion.CATATUMBO, 15000, 1186.0, "54245"),
2844:     PDETMunicipality("El Tarra", "Norte de Santander", PDETSubregion.CATATUMBO, 13000, 690.0, "54250"),
2845:     PDETMunicipality("HacarÃ-", "Norte de Santander", PDETSubregion.CATATUMBO, 14000, 549.0, "54344"),
2846:     PDETMunicipality("San Calixto", "Norte de Santander", PDETSubregion.CATATUMBO, 12000, 1155.0, "54660"),
2847:     PDETMunicipality("Sardinata", "Norte de Santander", PDETSubregion.CATATUMBO, 26000, 1398.0, "54720"),
2848:     PDETMunicipality("Teorama", "Norte de Santander", PDETSubregion.CATATUMBO, 19000, 1126.0, "54800"),
2849:     PDETMunicipality("TibÃº", "Norte de Santander", PDETSubregion.CATATUMBO, 48000, 2696.0, "54810"),
2850:
2851:     # CHOCÃ\223 (14 municipalities)
2852:     PDETMunicipality("AcandÃ-", "ChocÃ³", PDETSubregion.CHOCO, 11000, 993.0, "27006"),
2853:     PDETMunicipality("BojayÃ¡", "ChocÃ³", PDETSubregion.CHOCO, 11000, 1430.0, "27073"),
2854:     PDETMunicipality("Carmen del DariÃ©n", "ChocÃ³", PDETSubregion.CHOCO, 9000, 1995.0, "27135"),
2855:     PDETMunicipality("Condoto", "ChocÃ³", PDETSubregion.CHOCO, 20000, 1183.0, "27205"),
2856:     PDETMunicipality("Istmina", "ChocÃ³", PDETSubregion.CHOCO, 23000, 2394.0, "27361"),
2857:     PDETMunicipality("Litoral de San Juan", "ChocÃ³", PDETSubregion.CHOCO, 14000, 1024.0, "27413"),
2858:     PDETMunicipality("Medio Atrato", "ChocÃ³", PDETSubregion.CHOCO, 17000, 6815.0, "27425"),
2859:     PDETMunicipality("Medio San Juan", "ChocÃ³", PDETSubregion.CHOCO, 18000, 1331.0, "27430"),
2860:     PDETMunicipality("NÃ³vita", "ChocÃ³", PDETSubregion.CHOCO, 11000, 1619.0, "27491"),
2861:     PDETMunicipality("Riosucio", "ChocÃ³", PDETSubregion.CHOCO, 29000, 711.0, "27615"),
2862:     PDETMunicipality("SipÃ-", "ChocÃ³", PDETSubregion.CHOCO, 11000, 725.0, "27745"),
2863:     PDETMunicipality("UnguÃ-a", "ChocÃ³", PDETSubregion.CHOCO, 21000, 954.0, "27800"),
2864:     PDETMunicipality("MurindÃ³", "Antioquia", PDETSubregion.CHOCO, 4000, 1848.0, "05483"),
2865:     PDETMunicipality("VigÃ-a del Fuerte", "Antioquia", PDETSubregion.CHOCO, 6000, 956.0, "05873"),
2866:
2867:     # MACARENA-GUAVIARE (12 municipalities)
2868:     PDETMunicipality("MapiripÃ¡n", "Meta", PDETSubregion.MACARENA, 15000, 11341.0, "50325"),
2869:     PDETMunicipality("Mesetas", "Meta", PDETSubregion.MACARENA, 9000, 1430.0, "50330"),
2870:     PDETMunicipality("La Macarena", "Meta", PDETSubregion.MACARENA, 30000, 11229.0, "50350"),
2871:     PDETMunicipality("Uribe", "Meta", PDETSubregion.MACARENA, 15000, 9506.0, "50686"),
2872:     PDETMunicipality("Puerto Concordia", "Meta", PDETSubregion.MACARENA, 19000, 2077.0, "50568"),
2873:     PDETMunicipality("Puerto Lleras", "Meta", PDETSubregion.MACARENA, 12000, 3987.0, "50577"),
2874:     PDETMunicipality("Puerto Rico", "Meta", PDETSubregion.MACARENA, 19000, 2288.0, "50590"),
2875:     PDETMunicipality("Vista Hermosa", "Meta", PDETSubregion.MACARENA, 22000, 7417.0, "50711"),
2876:     PDETMunicipality("San JosÃ© del Guaviare", "Guaviare", PDETSubregion.MACARENA, 64000, 16592.0, "95001"),
2877:     PDETMunicipality("Calamar", "Guaviare", PDETSubregion.MACARENA, 23000, 36157.0, "95015"),
2878:     PDETMunicipality("El Retorno", "Guaviare", PDETSubregion.MACARENA, 19000, 18858.0, "95025"),
2879:     PDETMunicipality("Miraflores", "Guaviare", PDETSubregion.MACARENA, 8000, 27183.0, "95200"),
2880:
2881:     # MONTES DE MARÃ\215A (15 municipalities)
2882:     PDETMunicipality("CÃ³rdoba", "BolÃ-var", PDETSubregion.MONTES_MARIA, 14000, 336.0, "13212"),
2883:     PDETMunicipality("El Carmen de BolÃ-var", "BolÃ-var", PDETSubregion.MONTES_MARIA, 76000, 954.0, "13244"),
```

```
2884:     PDETMunicipality("El Guamo", "BolÃ-var", PDETSubregion.MONTES_MARIA, 10000, 179.0, "13268"),
2885:     PDETMunicipality("MarÃ-a la Baja", "BolÃ-var", PDETSubregion.MONTES_MARIA, 52000, 550.0, "13442"),
2886:     PDETMunicipality("San Jacinto", "BolÃ-var", PDETSubregion.MONTES_MARIA, 22000, 464.0, "13654"),
2887:     PDETMunicipality("San Juan Nepomuceno", "BolÃ-var", PDETSubregion.MONTES_MARIA, 39000, 547.0, "13657"),
2888:     PDETMunicipality("Zambrano", "BolÃ-var", PDETSubregion.MONTES_MARIA, 9000, 250.0, "13894"),
2889:     PDETMunicipality("ChalÃ¡n", "Sucre", PDETSubregion.MONTES_MARIA, 4000, 169.0, "70204"),
2890:     PDETMunicipality("Coloso", "Sucre", PDETSubregion.MONTES_MARIA, 6000, 237.0, "70215"),
2891:     PDETMunicipality("Los Palmitos", "Sucre", PDETSubregion.MONTES_MARIA, 21000, 321.0, "70429"),
2892:     PDETMunicipality("Morroa", "Sucre", PDETSubregion.MONTES_MARIA, 16000, 258.0, "70473"),
2893:     PDETMunicipality("Ovejas", "Sucre", PDETSubregion.MONTES_MARIA, 24000, 701.0, "70508"),
2894:     PDETMunicipality("Palmito", "Sucre", PDETSubregion.MONTES_MARIA, 12000, 126.0, "70523"),
2895:     PDETMunicipality("San Onofre", "Sucre", PDETSubregion.MONTES_MARIA, 50000, 1142.0, "70713"),
2896:     PDETMunicipality("TolÃ° Viejo", "Sucre", PDETSubregion.MONTES_MARIA, 25000, 231.0, "70823"),
2897:
2898:     # PACÃ\215FICO MEDIO (4 municipalities)
2899:     PDETMunicipality("Alto BaudÃ³", "ChocÃ³", PDETSubregion.PACIFICO_MEDIO, 35000, 1871.0, "27025"),
2900:     PDETMunicipality("Bajo BaudÃ³", "ChocÃ³", PDETSubregion.PACIFICO_MEDIO, 16000, 1862.0, "27050"),
2901:     PDETMunicipality("Medio BaudÃ³", "ChocÃ³", PDETSubregion.PACIFICO_MEDIO, 17000, 1803.0, "27420"),
2902:     PDETMunicipality("Buenaventura", "Valle del Cauca", PDETSubregion.PACIFICO_MEDIO, 424000, 6297.0, "76109"),
2903:
2904:     # PACÃ\215FICO Y FRONTERA NARIÃ\221ENSE (11 municipalities)
2905:     PDETMunicipality("Barbacoas", "NariÃ±o", PDETSubregion.PACIFICO_NARINENSE, 24000, 1674.0, "52083"),
2906:     PDETMunicipality("El Charco", "NariÃ±o", PDETSubregion.PACIFICO_NARINENSE, 32000, 2485.0, "52250"),
2907:     PDETMunicipality("Francisco Pizarro", "NariÃ±o", PDETSubregion.PACIFICO_NARINENSE, 13000, 1585.0, "52317"),
2908:     PDETMunicipality("La Tola", "NariÃ±o", PDETSubregion.PACIFICO_NARINENSE, 7000, 421.0, "52378"),
2909:     PDETMunicipality("MagÃ¼Ã- PayÃ¡n", "NariÃ±o", PDETSubregion.PACIFICO_NARINENSE, 23000, 1621.0, "52435"),
2910:     PDETMunicipality("Mosquera", "NariÃ±o", PDETSubregion.PACIFICO_NARINENSE, 12000, 1026.0, "52473"),
2911:     PDETMunicipality("Olaya Herrera", "NariÃ±o", PDETSubregion.PACIFICO_NARINENSE, 32000, 1932.0, "52490"),
2912:     PDETMunicipality("Roberto PayÃ¡n", "NariÃ±o", PDETSubregion.PACIFICO_NARINENSE, 18000, 1333.0, "52621"),
2913:     PDETMunicipality("Santa BÃ¡rbara", "NariÃ±o", PDETSubregion.PACIFICO_NARINENSE, 9000, 1398.0, "52683"),
2914:     PDETMunicipality("Tumaco", "NariÃ±o", PDETSubregion.PACIFICO_NARINENSE, 215000, 3760.0, "52835"),
2915:     PDETMunicipality("Ricaurte", "NariÃ±o", PDETSubregion.PACIFICO_NARINENSE, 16000, 505.0, "52612"),
2916:
2917:     # PUTUMAYO (9 municipalities)
2918:     PDETMunicipality("LeguÃ-zamo", "Putumayo", PDETSubregion.PUTUMAYO, 21000, 12421.0, "86573"),
2919:     PDETMunicipality("Mocoa", "Putumayo", PDETSubregion.PUTUMAYO, 46000, 1260.0, "86001"),
2920:     PDETMunicipality("Orito", "Putumayo", PDETSubregion.PUTUMAYO, 21000, 587.0, "86320"),
2921:     PDETMunicipality("Puerto AsÃ-s", "Putumayo", PDETSubregion.PUTUMAYO, 63000, 2961.0, "86568"),
2922:     PDETMunicipality("Puerto Caicedo", "Putumayo", PDETSubregion.PUTUMAYO, 16000, 1297.0, "86569"),
2923:     PDETMunicipality("Puerto GuzmÃ¡n", "Putumayo", PDETSubregion.PUTUMAYO, 17000, 3221.0, "86571"),
2924:     PDETMunicipality("San Miguel", "Putumayo", PDETSubregion.PUTUMAYO, 24000, 4086.0, "86755"),
2925:     PDETMunicipality("Valle del GuamuÃ©z", "Putumayo", PDETSubregion.PUTUMAYO, 49000, 1257.0, "86865"),
2926:     PDETMunicipality("VillagarzÃ³n", "Putumayo", PDETSubregion.PUTUMAYO, 18000, 1470.0, "86885"),
2927:
2928:     # SIERRA NEVADA - PERIJÃ\201 - ZONA BANANERA (15 municipalities)
2929:     PDETMunicipality("AgustÃ-n Codazzi", "Cesar", PDETSubregion.SIERRA_NEVADA, 62000, 2048.0, "20013"),
2930:     PDETMunicipality("Becerril", "Cesar", PDETSubregion.SIERRA_NEVADA, 18000, 690.0, "20045"),
2931:     PDETMunicipality("La Jagua de Ibirico", "Cesar", PDETSubregion.SIERRA_NEVADA, 22000, 720.0, "20383"),
2932:     PDETMunicipality("La Paz", "Cesar", PDETSubregion.SIERRA_NEVADA, 26000, 1238.0, "20400"),
2933:     PDETMunicipality("Manaure BalcÃ³n del Cesar", "Cesar", PDETSubregion.SIERRA_NEVADA, 15000, 1047.0, "20443"),
2934:     PDETMunicipality("Pueblo Bello", "Cesar", PDETSubregion.SIERRA_NEVADA, 14000, 612.0, "20570"),
2935:     PDETMunicipality("San Diego", "Cesar", PDETSubregion.SIERRA_NEVADA, 14000, 474.0, "20621"),
2936:     PDETMunicipality("Valledupar", "Cesar", PDETSubregion.SIERRA_NEVADA, 490000, 4493.0, "20001"),
2937:     PDETMunicipality("Dibulla", "La Guajira", PDETSubregion.SIERRA_NEVADA, 33000, 1774.0, "44090"),
2938:     PDETMunicipality("Fonseca", "La Guajira", PDETSubregion.SIERRA_NEVADA, 36000, 494.0, "44279"),
2939:     PDETMunicipality("San Juan del Cesar", "La Guajira", PDETSubregion.SIERRA_NEVADA, 40000, 1671.0, "44650"),
```

```
2940:     PDETMunicipality("Aracataca", "Magdalena", PDETSubregion.SIERRA_NEVADA, 42000, 1254.0, "47053"),
2941:     PDETMunicipality("CiÃ©naga", "Magdalena", PDETSubregion.SIERRA_NEVADA, 104000, 1237.0, "47189"),
2942:     PDETMunicipality("FundaciÃ³n", "Magdalena", PDETSubregion.SIERRA_NEVADA, 63000, 988.0, "47288"),
2943:     PDETMunicipality("Santa Marta", "Magdalena", PDETSubregion.SIERRA_NEVADA, 500000, 2381.0, "47001"),
2944:
2945:     # SUR DE BOLÃ\215VAR (7 municipalities)
2946:     PDETMunicipality("Arenal", "BolÃ-var", PDETSubregion.SUR_BOLIVAR, 18000, 627.0, "13052"),
2947:     PDETMunicipality("Cantagallo", "BolÃ-var", PDETSubregion.SUR_BOLIVAR, 12000, 1202.0, "13140"),
2948:     PDETMunicipality("Morales", "BolÃ-var", PDETSubregion.SUR_BOLIVAR, 17000, 416.0, "13468"),
2949:     PDETMunicipality("San Pablo", "BolÃ-var", PDETSubregion.SUR_BOLIVAR, 44000, 979.0, "13667"),
2950:     PDETMunicipality("Santa Rosa del Sur", "BolÃ-var", PDETSubregion.SUR_BOLIVAR, 39000, 1749.0, "13688"),
2951:     PDETMunicipality("SimitÃ-", "BolÃ-var", PDETSubregion.SUR_BOLIVAR, 20000, 2814.0, "13744"),
2952:     PDETMunicipality("YondÃ³", "Antioquia", PDETSubregion.SUR_BOLIVAR, 18000, 1635.0, "05893"),
2953:
2954:     # SUR DE CÃ\223RDOBA (5 municipalities)
2955:     PDETMunicipality("MontelÃ-bano", "CÃ³rdoba", PDETSubregion.SUR_CORDOBA, 83000, 2515.0, "23466"),
2956:     PDETMunicipality("Puerto Libertador", "CÃ³rdoba", PDETSubregion.SUR_CORDOBA, 39000, 2903.0, "23570"),
2957:     PDETMunicipality("San JosÃ© de UrÃ©", "CÃ³rdoba", PDETSubregion.SUR_CORDOBA, 12000, 1298.0, "23682"),
2958:     PDETMunicipality("Tierralta", "CÃ³rdoba", PDETSubregion.SUR_CORDOBA, 101000, 5084.0, "23807"),
2959:     PDETMunicipality("Valencia", "CÃ³rdoba", PDETSubregion.SUR_CORDOBA, 41000, 752.0, "23855"),
2960:
2961:     # SUR DEL TOLIMA (4 municipalities)
2962:     PDETMunicipality("Ataco", "Tolima", PDETSubregion.SUR_TOLIMA, 23000, 554.0, "73067"),
2963:     PDETMunicipality("Chaparral", "Tolima", PDETSubregion.SUR_TOLIMA, 48000, 2238.0, "73168"),
2964:     PDETMunicipality("Planadas", "Tolima", PDETSubregion.SUR_TOLIMA, 30000, 908.0, "73547"),
2965:     PDETMunicipality("Rioblanco", "Tolima", PDETSubregion.SUR_TOLIMA, 23000, 1352.0, "73616"),
2966:
2967:     # URABÃ\201 ANTIOQUEÃ\221O (10 municipalities)
2968:     PDETMunicipality("ApartadÃ³", "Antioquia", PDETSubregion.URABA, 195000, 607.0, "05045"),
2969:     PDETMunicipality("Carepa", "Antioquia", PDETSubregion.URABA, 58000, 197.0, "05147"),
2970:     PDETMunicipality("ChigorodÃ³", "Antioquia", PDETSubregion.URABA, 79000, 615.0, "05172"),
2971:     PDETMunicipality("MutatÃ¡", "Antioquia", PDETSubregion.URABA, 20000, 1185.0, "05490"),
2972:     PDETMunicipality("NecoclÃ-", "Antioquia", PDETSubregion.URABA, 66000, 1387.0, "05490"),
2973:     PDETMunicipality("San Juan de UrabÃ¡", "Antioquia", PDETSubregion.URABA, 23000, 672.0, "05659"),
2974:     PDETMunicipality("San Pedro de UrabÃ¡", "Antioquia", PDETSubregion.URABA, 37000, 401.0, "05664"),
2975:     PDETMunicipality("Turbo", "Antioquia", PDETSubregion.URABA, 165000, 3055.0, "05837"),
2976:     PDETMunicipality("Arboletes", "Antioquia", PDETSubregion.URABA, 40000, 647.0, "05051"),
2977:     PDETMunicipality("Dabeiba", "Antioquia", PDETSubregion.URABA, 25000, 1256.0, "05234"),
2978: ]
2979:
2980:
2981: def get_municipalities_by_subregion(subregion: PDETSubregion) -> list[PDETMunicipality]:
2982:     """Get all municipalities for a specific subregion"""
2983:     return [m for m in PDET_MUNICIPALITIES if m.subregion == subregion]
2984:
2985:
2986: def get_municipalities_by_department(department: str) -> list[PDETMunicipality]:
2987:     """Get all municipalities for a specific department"""
2988:     return [m for m in PDET_MUNICIPALITIES if m.department == department]
2989:
2990:
2991: def get_municipality_by_name(name: str) -> PDETMunicipality:
2992:     """Get municipality by name"""
2993:     for m in PDET_MUNICIPALITIES:
2994:         if m.name.lower() == name.lower():
2995:             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"



================================================================================
FILE: src/farfan_pipeline/api/pipeline_connector.py
================================================================================

"""
AtroZ Pipeline Connector
Real integration with the orchestrator for executing the 11-phase analysis pipeline
"""

import json
import logging
import time
import traceback
from collections.abc import Callable
from dataclasses import asdict, dataclass
from datetime import datetime
from pathlib import Path
from typing import Any

from farfan_pipeline.core.calibration.decorators import calibrated_method
from farfan_pipeline.core.orchestrator.core import Orchestrator
from farfan_pipeline.core.orchestrator.factory import build_processor
from farfan_pipeline.core.orchestrator.questionnaire import load_questionnaire
from farfan_pipeline.core.orchestrator.verification_manifest import VerificationManifest

logger = logging.getLogger(__name__)


```

```
@dataclass
class PipelineResult:
    """Complete result from pipeline execution"""
    success: bool
    job_id: str
    document_id: str
    duration_seconds: float
    phases_completed: int
    macro_score: float | None
    meso_scores: dict[str, float] | None
    micro_scores: dict[str, float] | None
    questions_analyzed: int
    evidence_count: int
    recommendations_count: int
    verification_manifest_path: str | None
    error: str | None
    phase_timings: dict[str, float]
    metadata: dict[str, Any]


class PipelineConnector:
    """
    Connector for executing the real F.A.R.F.A.N pipeline through the Orchestrator.

    This class provides the bridge between the API layer and the core analysis engine,
    handling document ingestion, pipeline execution, progress tracking, and result extraction.
    """

    def __init__(self, workspace_dir: str | None = None, output_dir: str | None = None) -> None:
        """Initialize PipelineConnector with centralized path management.

        Args:
            workspace_dir: Optional workspace directory (defaults to paths.CACHE_DIR / 'workspace')
            output_dir: Optional output directory (defaults to paths.OUTPUT_DIR)
        """
        from farfan_pipeline.config.paths import CACHE_DIR, OUTPUT_DIR, ensure_directories_exist

        # Use centralized paths by default
        if workspace_dir is None:
            self.workspace_dir = CACHE_DIR / 'workspace'
        else:
            self.workspace_dir = Path(workspace_dir)

        if output_dir is None:
            self.output_dir = OUTPUT_DIR
        else:
            self.output_dir = Path(output_dir)

        # Ensure directories exist
        ensure_directories_exist()
        self.workspace_dir.mkdir(parents=True, exist_ok=True)
        self.output_dir.mkdir(parents=True, exist_ok=True)

        self.running_jobs: dict[str, dict[str, Any]] = {}
        self.completed_jobs: dict[str, PipelineResult] = {}

```

```
3108:         logger.info(
3109:             f"Pipeline connector initialized with centralized paths: "
3110:             f"workspace={self.workspace_dir}, output={self.output_dir}"
3111:         )
3112:
3113:     async def execute_pipeline(
3114:         self,
3115:         pdf_path: str,
3116:         job_id: str,
3117:         municipality: str = "general",
3118:         progress_callback: Callable[[int, str], None] | None = None,
3119:         settings: dict[str, Any] | None = None
3120:     ) -> PipelineResult:
3121:         """
3122:         Execute the complete 11-phase pipeline on a PDF document.
3123:
3124:         Args:
3125:             pdf_path: Path to the PDF document to analyze
3126:             job_id: Unique identifier for this job
3127:             municipality: Municipality name for context
3128:             progress_callback: Optional callback function(phase_num, phase_name) for progress updates
3129:             settings: Optional pipeline settings (timeout, cache, etc.)
3130:
3131:         Returns:
3132:             PipelineResult with complete analysis results
3133:         """
3134:         start_time = time.time()
3135:         settings = settings or {}
3136:
3137:         logger.info(f"Starting pipeline execution for job {job_id}: {pdf_path}")
3138:
3139:         self.running_jobs[job_id] = {
3140:             "status": "initializing",
3141:             "start_time": start_time,
3142:             "current_phase": None,
3143:             "progress": 0
3144:         }
3145:
3146:         try:
3147:             # Phase 0: Document Ingestion
3148:             if progress_callback:
3149:                 progress_callback(0, "Ingesting document")
3150:             self._update_job_status(job_id, "ingesting", 0, "Document ingestion")
3151:
3152:             preprocessed_doc = await self._ingest_document(pdf_path, municipality)
3153:
3154:             # Initialize Orchestrator with proper factory and questionnaire
3155:             # FIX: Previously used Orchestrator() without parameters which would fail
3156:             # in _load_configuration() with ValueError: "No monolith data available"
3157:             logger.info("Initializing Orchestrator via factory pattern")
3158:
3159:             # Build processor bundle with all dependencies
3160:             processor = build_processor()
3161:
3162:             # Load canonical questionnaire for type-safe initialization
3163:             canonical_questionnaire = load_questionnaire()
```

```
3164:
3165:             # Initialize orchestrator with pre-loaded data (I/O-free path)
3166:             orchestrator = Orchestrator(
3167:                 questionnaire=canonical_questionnaire,
3168:                 catalog=processor.factory.catalog
3169:             )
3170:
3171:             logger.info(
3172:                 "Orchestrator initialized successfully",
3173:                 extra={
3174:                     "questionnaire_hash": canonical_questionnaire.sha256[:16] + "...",
3175:                     "question_count": canonical_questionnaire.total_question_count,
3176:                     "catalog_loaded": processor.factory.catalog is not None
3177:                 }
3178:             )
3179:
3180:             # Track phase timings
3181:             phase_timings = {}
3182:             phase_start_times = {}
3183:
3184:             # Define progress callback for real-time updates from orchestrator
3185:             def orchestrator_progress_callback(phase_num: int, phase_name: str, progress: float) -> None:
3186:                 """Callback invoked by orchestrator for each phase.
3187:
3188:                 Args:
3189:                     phase_num: Phase number (0-10)
3190:                     phase_name: Phase name
3191:                     progress: Progress percentage (0-100)
3192:                 """
3193:                 # Track phase timing
3194:                 if phase_num not in phase_start_times:
3195:                     phase_start_times[phase_num] = time.time()
3196:                 else:
3197:                     # Phase complete - record duration
3198:                     duration = time.time() - phase_start_times[phase_num]
3199:                     phase_timings[f"phase_{phase_num}"] = duration
3200:
3201:                 # Update job status
3202:                 self._update_job_status(job_id, "processing", int(progress), phase_name)
3203:
3204:                 # Call user's progress callback if provided
3205:                 if progress_callback:
3206:                     progress_callback(phase_num, phase_name)
3207:
3208:                 logger.info(
3209:                     f"Orchestrator Phase {phase_num}: {phase_name} ({progress:.1f}% complete)"
3210:                 )
3211:
3212:             # Run the complete orchestrator with real phase callbacks
3213:             logger.info("Running complete orchestrator pipeline with real-time progress")
3214:             orchestrator_start = time.time()
3215:
3216:             result = await orchestrator.run(
3217:                 preprocessed_doc=preprocessed_doc,
3218:                 output_path=str(self.output_dir / f"{job_id}_report.json"),
3219:                 phase_timeout=settings.get("phase_timeout", 300),
```

```
3220:                 enable_cache=settings.get("enable_cache", True),
3221:                 progress_callback=orchestrator_progress_callback,
3222:             )
3223:
3224:             orchestrator_duration = time.time() - orchestrator_start
3225:             logger.info(f"Orchestrator completed in {orchestrator_duration:.2f}s")
3226:
3227:             # Extract metrics from result
3228:             metrics = self._extract_metrics(result)
3229:
3230:             # Write verification manifest
3231:             manifest_path = await self._write_manifest(job_id, result, metrics)
3232:
3233:             # Create result object
3234:             pipeline_result = PipelineResult(
3235:                 success=True,
3236:                 job_id=job_id,
3237:                 document_id=preprocessed_doc.get("document_id", job_id),
3238:                 duration_seconds=time.time() - start_time,
3239:                 phases_completed=11,
3240:                 macro_score=metrics.get("macro_score"),
3241:                 meso_scores=metrics.get("meso_scores"),
3242:                 micro_scores=metrics.get("micro_scores"),
3243:                 questions_analyzed=metrics.get("questions_analyzed", 0),
3244:                 evidence_count=metrics.get("evidence_count", 0),
3245:                 recommendations_count=metrics.get("recommendations_count", 0),
3246:                 verification_manifest_path=manifest_path,
3247:                 error=None,
3248:                 phase_timings=phase_timings,
3249:                 metadata={
3250:                     "municipality": municipality,
3251:                     "pdf_path": pdf_path,
3252:                     "orchestrator_version": result.get("version", "unknown"),
3253:                     "completed_at": datetime.now().isoformat()
3254:                 }
3255:             )
3256:
3257:             self.completed_jobs[job_id] = pipeline_result
3258:             self._update_job_status(job_id, "completed", 100, "Analysis complete")
3259:
3260:             logger.info(f"Pipeline execution completed successfully for job {job_id}")
3261:             return pipeline_result
3262:
3263:         except Exception as e:
3264:             error_msg = f"Pipeline execution failed: {str(e)}"
3265:             logger.error(f"{error_msg}\n{traceback.format_exc()}")
3266:
3267:             pipeline_result = PipelineResult(
3268:                 success=False,
3269:                 job_id=job_id,
3270:                 document_id="unknown",
3271:                 duration_seconds=time.time() - start_time,
3272:                 phases_completed=0,
3273:                 macro_score=None,
3274:                 meso_scores=None,
3275:                 micro_scores=None,
```

```
                questions_analyzed=0,
                evidence_count=0,
                recommendations_count=0,
                verification_manifest_path=None,
                error=error_msg,
                phase_timings={},
                metadata={"error_traceback": traceback.format_exc()}
            )

            self.completed_jobs[job_id] = pipeline_result
            self._update_job_status(job_id, "failed", 0, error_msg)

            return pipeline_result

        finally:
            if job_id in self.running_jobs:
                del self.running_jobs[job_id]

    async def _ingest_document(self, pdf_path: str, municipality: str) -> Any:
        """
        Ingest and preprocess the PDF document using canonical SPC pipeline.

        This method implements the official ingestion path:
            CPPIngestionPipeline â\206\222 CPPAdapter â\206\222 PreprocessedDocument

        Args:
            pdf_path: Path to PDF document
            municipality: Municipality name for metadata

        Returns:
            PreprocessedDocument ready for orchestrator

        Raises:
            ValueError: If ingestion fails or produces invalid output

        Note:
            Uses factory methods for dependency injection instead of direct imports.
        """
        from pathlib import Path

        from farfan_pipeline.core.orchestrator.factory import (
            create_cpp_adapter,
            create_cpp_ingestion_pipeline,
        )

        logger.info(f"Ingesting document via canonical SPC pipeline: {pdf_path}")

        try:
            # Phase 1: CPP Ingestion (15-phase SPC analysis) - via factory
            cpp_pipeline = create_cpp_ingestion_pipeline(enable_runtime_validation=True)

            document_path = Path(pdf_path)
            if not document_path.exists():
                raise ValueError(f"Document not found: {pdf_path}")

            # Generate document_id from filename and timestamp for uniqueness
```

```
3332:             document_id = f"{document_path.stem}_{int(time.time())}"
3333:             title = f"{municipality} - {document_path.name}"
3334:
3335:             logger.info("Running CPPIngestionPipeline (15-phase SPC analysis)")
3336:             canon_package = await cpp_pipeline.process(
3337:                 document_path=document_path,
3338:                 document_id=document_id,
3339:                 title=title,
3340:                 max_chunks=50,
3341:             )
3342:
3343:             logger.info(
3344:                 f"CPP Ingestion complete: {len(canon_package.chunk_graph.chunks)} chunks generated"
3345:             )
3346:
3347:             # Phase 2: CPP Adapter (convert to PreprocessedDocument) - via factory
3348:             adapter = create_cpp_adapter(enable_runtime_validation=True)
3349:
3350:             logger.info("Converting CanonPolicyPackage to PreprocessedDocument")
3351:             preprocessed_doc = adapter.to_preprocessed_document(
3352:                 canon_package=canon_package,
3353:                 document_id=document_id,
3354:             )
3355:
3356:             logger.info(
3357:                 f"Adapter conversion complete: {len(preprocessed_doc.sentences)} sentences"
3358:             )
3359:
3360:             # Add municipality to metadata if needed
3361:             if hasattr(preprocessed_doc, 'metadata') and isinstance(preprocessed_doc.metadata, dict):
3362:                 # metadata is MappingProxyType (immutable), so we need to create a new one
3363:                 from types import MappingProxyType
3364:                 metadata_dict = dict(preprocessed_doc.metadata)
3365:                 metadata_dict['municipality'] = municipality
3366:                 metadata_dict['source_path'] = str(pdf_path)
3367:
3368:                 # Reconstruct PreprocessedDocument with updated metadata
3369:                 preprocessed_doc = type(preprocessed_doc)(
3370:                     document_id=preprocessed_doc.document_id,
3371:                     full_text=preprocessed_doc.full_text,
3372:                     sentences=preprocessed_doc.sentences,
3373:                     language=preprocessed_doc.language,
3374:                     structured_text=preprocessed_doc.structured_text,
3375:                     sentence_metadata=preprocessed_doc.sentence_metadata,
3376:                     tables=preprocessed_doc.tables,
3377:                     indexes=preprocessed_doc.indexes,
3378:                     metadata=MappingProxyType(metadata_dict),
3379:                     ingested_at=preprocessed_doc.ingested_at,
3380:                 )
3381:
3382:             return preprocessed_doc
3383:
3384:         except Exception as e:
3385:             logger.error(f"Canonical ingestion failed: {e}", exc_info=True)
3386:             raise ValueError(
3387:                 f"Document ingestion failed for {pdf_path}: {e}\n"
```

```
3388:                 f"Ensure CPPIngestionPipeline and SPCAdapter are working correctly."
3389:             ) from e
3390:
3391:     @calibrated_method("farfan_core.api.pipeline_connector.PipelineConnector._extract_metrics")
3392:     def _extract_metrics(self, orchestrator_result: dict[str, Any]) -> dict[str, Any]:
3393:         """Extract key metrics from orchestrator result"""
3394:         metrics = {}
3395:
3396:         # Extract macro score
3397:         if "macro_analysis" in orchestrator_result:
3398:             macro_data = orchestrator_result["macro_analysis"]
3399:             metrics["macro_score"] = macro_data.get("overall_score")
3400:
3401:         # Extract meso scores
3402:         if "meso_analysis" in orchestrator_result:
3403:             meso_data = orchestrator_result["meso_analysis"]
3404:             metrics["meso_scores"] = meso_data.get("cluster_scores", {})
3405:
3406:         # Extract micro scores
3407:         if "micro_analysis" in orchestrator_result:
3408:             micro_data = orchestrator_result["micro_analysis"]
3409:             metrics["micro_scores"] = micro_data.get("question_scores", {})
3410:             metrics["questions_analyzed"] = len(micro_data.get("questions", []))
3411:             metrics["evidence_count"] = sum(
3412:                 len(q.get("evidence", []))
3413:                 for q in micro_data.get("questions", [])
3414:             )
3415:
3416:         # Extract recommendations
3417:         if "recommendations" in orchestrator_result:
3418:             metrics["recommendations_count"] = len(orchestrator_result["recommendations"])
3419:
3420:         return metrics
3421:
3422:     async def _write_manifest(
3423:         self,
3424:         job_id: str,
3425:         orchestrator_result: dict[str, Any],
3426:         metrics: dict[str, Any]
3427:     ) -> str:
3428:         """Write verification manifest for the analysis using centralized paths."""
3429:         from farfan_pipeline.config.paths import get_output_path
3430:
3431:         # Use centralized path management for manifest
3432:         job_output_dir = get_output_path(job_id)
3433:         manifest_path = job_output_dir / "verification_manifest.json"
3434:
3435:         manifest_data = {
3436:             "job_id": job_id,
3437:             "timestamp": datetime.now().isoformat(),
3438:             "status": "completed",
3439:             "metrics": metrics,
3440:             "verification": {
3441:                 "phases_completed": orchestrator_result.get("phases_completed", 11),
3442:                 "data_integrity": "verified",
3443:                 "output_path": str(job_output_dir / "report.json"),
```

```
                "wiring_validated": True,  # Runtime wiring validation enabled
            },
            "pipeline_metadata": {
                "spc_pipeline": "CPPIngestionPipeline",
                "adapter": "SPCAdapter",
                "orchestrator_version": orchestrator_result.get("metadata", {}).get("orchestrator_version", "2.0"),
            }
        }

        try:
            manifest = VerificationManifest()
            manifest.set_success(True)
            manifest.manifest_data.update(manifest_data)
            manifest.write(str(manifest_path))
        except Exception as e:
            logger.warning(f"Could not write verification manifest: {e}")
            with open(manifest_path, 'w', encoding='utf-8') as f:
                json.dump(manifest_data, f, indent=2, ensure_ascii=False)

        logger.info(f"Verification manifest written to: {manifest_path}")
        return str(manifest_path)

    @calibrated_method("farfan_core.api.pipeline_connector.PipelineConnector._update_job_status")
    def _update_job_status(self, job_id: str, status: str, progress: int, message: str) -> None:
        """Update status of running job"""
        if job_id in self.running_jobs:
            self.running_jobs[job_id].update({
                "status": status,
                "progress": progress,
                "current_phase": message,
                "updated_at": datetime.now().isoformat()
            })

    @calibrated_method("farfan_core.api.pipeline_connector.PipelineConnector.get_job_status")
    def get_job_status(self, job_id: str) -> dict[str, Any] | None:
        """Get current status of a job"""
        if job_id in self.running_jobs:
            return self.running_jobs[job_id]
        elif job_id in self.completed_jobs:
            result = self.completed_jobs[job_id]
            return {
                "status": "completed" if result.success else "failed",
                "progress": 100 if result.success else 0,
                "result": asdict(result)
            }
        return None

    @calibrated_method("farfan_core.api.pipeline_connector.PipelineConnector.get_result")
    def get_result(self, job_id: str) -> PipelineResult | None:
        """Get final result for a completed job"""
        return self.completed_jobs.get(job_id)


# Global connector instance
_connector: PipelineConnector | None = None

```

```
3500:
3501: def get_pipeline_connector() -> PipelineConnector:
3502:     """Get or create global pipeline connector instance"""
3503:     global _connector
3504:     if _connector is None:
3505:         _connector = PipelineConnector()
3506:     return _connector
3507:
3508:
3509:
3510: ================================================================================
3511: FILE: src/farfan_pipeline/api/signals_service.py
3512: ================================================================================
3513:
3514: """FastAPI Signal Service - Cross-Cut Channel Publisher.
3515:
3516: This service exposes signal packs from questionnaire.monolith to the orchestrator
3517: via HTTP endpoints with ETag support, caching, and SSE streaming.
3518:
3519: Endpoints:
3520: - GET /signals/{policy_area}: Fetch signal pack for policy area
3521: - GET /signals/stream: SSE stream of signal updates
3522: - GET /health: Health check endpoint
3523:
3524: Design:
3525: - ETag support for efficient cache invalidation
3526: - Cache-Control headers for client-side caching
3527: - SSE for real-time signal updates
3528: - OpenTelemetry instrumentation
3529: - Structured logging
3530: """
3531:
3532: from __future__ import annotations
3533:
3534: import asyncio
3535: import json
3536: from datetime import datetime, timezone
3537: from typing import TYPE_CHECKING
3538:
3539: import structlog
3540: from fastapi import FastAPI, HTTPException, Request, Response
3541: from sse_starlette.sse import EventSourceResponse
3542:
3543: from farfan_pipeline.core.orchestrator.questionnaire import load_questionnaire
3544: from farfan_pipeline.core.orchestrator.signals import PolicyArea, SignalPack
3545: from farfan_pipeline.core.calibration.decorators import calibrated_method
3546:
3547: if TYPE_CHECKING:
3548:     from collections.abc import AsyncIterator
3549:     from pathlib import Path
3550:
3551: logger = structlog.get_logger(__name__)
3552:
3553:
3554: # In-memory signal store (would be database/file in production)
3555: _signal_store: dict[str, SignalPack] = {}
```

```
3556:
3557:
3558: def load_signals_from_monolith(monolith_path: str | Path | None = None) -> dict[str, SignalPack]:
3559:     """
3560:     Load signal packs from questionnaire monolith using canonical loader.
3561:
3562:     Uses questionnaire.load_questionnaire() for hash verification and immutability.
3563:     This extracts policy-aware patterns, indicators, and thresholds from the
3564:     questionnaire monolith and converts them into SignalPack format.
3565:
3566:     Args:
3567:         monolith_path: DEPRECATED - Path parameter is ignored.
3568:                        Questionnaire always loads from canonical path.
3569:
3570:     Returns:
3571:         Dict mapping policy area to SignalPack
3572:
3573:     TODO: Implement actual extraction logic from monolith structure
3574:     """
3575:     if monolith_path is not None:
3576:         logger.info(
3577:             "monolith_path_ignored",
3578:             provided_path=str(monolith_path),
3579:             message="Path parameter ignored. Using canonical loader.",
3580:         )
3581:
3582:     try:
3583:         # Use canonical loader (no path parameter - always canonical path)
3584:         canonical_q = load_questionnaire()
3585:
3586:         logger.info(
3587:             "signals_loaded_from_monolith",
3588:             path=str(monolith_path),
3589:             sha256=canonical_q.sha256[:16] + "...",
3590:             question_count=canonical_q.total_question_count,
3591:             message="TODO: Implement actual extraction",
3592:         )
3593:
3594:         # TODO: Implement extraction logic using canonical_q.data
3595:         return _create_stub_signal_packs()
3596:
3597:     except Exception as e:
3598:         logger.error("failed_to_load_monolith", path=str(monolith_path), error=str(e))
3599:         return _create_stub_signal_packs()
3600:
3601:
3602: def _create_stub_signal_packs() -> dict[str, SignalPack]:
3603:     """Create stub signal packs for all policy areas."""
3604:     policy_areas: list[PolicyArea] = [
3605:         "fiscal",
3606:         "salud",
3607:         "ambiente",
3608:         "energÃ-a",
3609:         "transporte",
3610:     ]
3611:
```

```
    packs = {}
    for area in policy_areas:
        packs[area] = SignalPack(
            version="1.0.0",
            policy_area=area,
            patterns=[
                f"patrÃ³n_{area}_1",
                f"patrÃ³n_{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.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)
```

```
3668:
3669:     logger.info(
3670:         "signal_service_started",
3671:         signal_count=len(_signal_store),
3672:         policy_areas=list(_signal_store.keys()),
3673:     )
3674:
3675:
3676: @app.get("/health")
3677: async def health_check() -> dict[str, str]:
3678:     """
3679:     Health check endpoint.
3680:
3681:     Returns:
3682:         Status dict
3683:     """
3684:     return {
3685:         "status": "healthy",
3686:         "timestamp": datetime.now(timezone.utc).isoformat(),
3687:         "signal_count": len(_signal_store),
3688:     }
3689:
3690:
3691: @app.get("/signals/{policy_area}")
3692: async def get_signal_pack(
3693:     policy_area: str,
3694:     request: Request,
3695:     response: Response,
3696: ) -> SignalPack:
3697:     """
3698:     Fetch signal pack for a policy area.
3699:
3700:     Supports:
3701:     - ETag-based caching
3702:     - Cache-Control headers
3703:     - Conditional requests (If-None-Match)
3704:
3705:     Args:
3706:         policy_area: Policy area identifier
3707:         request: FastAPI request
3708:         response: FastAPI response
3709:
3710:     Returns:
3711:         SignalPack for the requested policy area
3712:
3713:     Raises:
3714:         HTTPException: If policy area not found
3715:     """
3716:     # Validate policy area
3717:     if policy_area not in _signal_store:
3718:         logger.warning("signal_pack_not_found", policy_area=policy_area)
3719:         raise HTTPException(status_code=404, detail=f"Policy area '{policy_area}' not found")
3720:
3721:     signal_pack = _signal_store[policy_area]
3722:
3723:     # 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_core.api.signals_service:app",
        host="0.0.0.0",
        port=8000,
        log_level="info",
        reload=False,
    )


if __name__ == "__main__":
    main()



================================================================================
FILE: src/farfan_pipeline/api/static/__init__.py
================================================================================

"""API static files."""



================================================================================
FILE: src/farfan_pipeline/api/static/js/__init__.py
================================================================================

"""API static JavaScript files."""



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

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

Comprehensive audit system for verifying architectural compliance,
dependency injection patterns, and code quality standards.

Author: FARFAN Team
Date: 2025-11-13
```

```
Version: 1.0.0
"""

from farfan_pipeline.audit.audit_system import (
    AuditCategory,
    AuditFinding,
    AuditStatus,
    AuditSystem,
    ExecutorAuditInfo,
)

__all__ = [
    "AuditCategory",
    "AuditFinding",
    "AuditStatus",
    "AuditSystem",
    "ExecutorAuditInfo",
]


================================================================================
FILE: src/farfan_pipeline/audit/audit_system.py
================================================================================

"""
FARFAN Mechanistic Policy Pipeline - Audit System
==================================================

This module provides comprehensive audit capabilities to verify:
1. Executor Architecture (30 dimension-question executors)
2. Questionnaire Access Patterns (dependency injection only)
3. Factory Pattern Compliance
4. Method Signature Completeness
5. Configuration System Type-Safety

AUDIT COMPLIANCE MARKERS:
- â\234\205 AUDIT_VERIFIED: Component passes all audit checks
- â\232 ï¸\217  AUDIT_WARNING: Component has potential issues
- â\235\214 AUDIT_FAILED: Component fails audit requirements

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

import ast
import json
import logging
import sys
from dataclasses import dataclass, field
from datetime import datetime
from enum import Enum
from pathlib import Path
from typing import Any

```

```
3948: from farfan_pipeline.config.paths import PROJECT_ROOT
3949: from farfan_pipeline.core.canonical_notation import CanonicalDimension
3950:
3951: logger = logging.getLogger(__name__)
3952:
3953:
3954: class AuditStatus(Enum):
3955:     """Audit status enumeration."""
3956:     VERIFIED = "â\234\205 VERIFIED"
3957:     WARNING = "â\232 ï¸\217  WARNING"
3958:     FAILED = "â\235\214 FAILED"
3959:
3960:
3961: class AuditCategory(Enum):
3962:     """Audit category enumeration."""
3963:     EXECUTOR_ARCHITECTURE = "Executor Architecture"
3964:     QUESTIONNAIRE_ACCESS = "Questionnaire Access"
3965:     FACTORY_PATTERN = "Factory Pattern"
3966:     METHOD_SIGNATURES = "Method Signatures"
3967:     CONFIGURATION_SYSTEM = "Configuration System"
3968:     SAGA_PATTERN = "Saga Pattern"
3969:     EVENT_TRACKING = "Event Tracking"
3970:     RL_OPTIMIZATION = "RL Optimization"
3971:     INTEGRATION_TESTS = "Integration Tests"
3972:     OBSERVABILITY = "Observability"
3973:
3974:
3975: @dataclass
3976: class AuditFinding:
3977:     """Represents a single audit finding."""
3978:     category: AuditCategory
3979:     status: AuditStatus
3980:     component: str
3981:     message: str
3982:     details: dict[str, Any] = field(default_factory=dict)
3983:     timestamp: datetime = field(default_factory=datetime.utcnow)
3984:
3985:     def to_dict(self) -> dict[str, Any]:
3986:         """Convert finding to dictionary."""
3987:         return {
3988:             "category": self.category.value,
3989:             "status": self.status.value,
3990:             "component": self.component,
3991:             "message": self.message,
3992:             "details": self.details,
3993:             "timestamp": self.timestamp.isoformat()
3994:         }
3995:
3996:     def __str__(self) -> str:
3997:         """String representation of finding."""
3998:         return f"{self.status.value} [{self.category.value}] {self.component}: {self.message}"
3999:
4000:
4001: @dataclass
4002: class ExecutorAuditInfo:
4003:     """Information about an executor for audit purposes."""
```

```
    executor_name: str
    dimension: int  # 1-6
    question: int  # 1-5
    dimension_name: str
    class_exists: bool
    has_execute_method: bool
    accesses_questionnaire_directly: bool
    uses_dependency_injection: bool
    file_path: str | None = None
    line_number: int | None = None


class AuditSystem:
    """
    Comprehensive audit system for FARFAN Pipeline.

    This class provides methods to audit all critical components of the pipeline
    to ensure compliance with architectural requirements.
    """

    # Expected 30 executors (D1Q1-D6Q5)
    EXPECTED_EXECUTORS = [
        f"D{d}Q{q}_Executor"
        for d in range(1, 7)
        for q in range(1, 6)
    ]

    # Dimension names from canonical notation
    DIMENSION_NAMES = {
        idx: getattr(CanonicalDimension, f"D{idx}").label
        for idx in range(1, 7)
    }

    # Core scripts that MUST use dependency injection
    CORE_SCRIPTS = [
        "policy_processor.py",
        "Analyzer_one.py",
        "embedding_policy.py",
        "financiero_viabilidad_tablas.py",
        "teoria_cambio.py",
        "dereck_beach.py",
        "semantic_chunking_policy.py"
    ]

    def __init__(self, repo_root: Path) -> None:
        """
        Initialize audit system.

        Args:
            repo_root: Root directory of the repository
        """
        self.repo_root = repo_root
        self.findings: list[AuditFinding] = []

    def add_finding(
        self,
```

```
4060:         category: AuditCategory,
4061:         status: AuditStatus,
4062:         component: str,
4063:         message: str,
4064:         details: dict[str, Any] | None = None
4065:     ) -> None:
4066:         """Add an audit finding."""
4067:         finding = AuditFinding(
4068:             category=category,
4069:             status=status,
4070:             component=component,
4071:             message=message,
4072:             details=details or {}
4073:         )
4074:         self.findings.append(finding)
4075:         logger.info(str(finding))
4076:
4077:     def audit_executor_architecture(self) -> dict[str, Any]:
4078:         """
4079:         Audit the 30-executor architecture (D1Q1-D6Q5).
4080:
4081:         Returns:
4082:             Dictionary with audit results
4083:         """
4084:         logger.info("=" * 80)
4085:         logger.info("AUDITING: Executor Architecture (30 Dimension-Question Executors)")
4086:         logger.info("=" * 80)
4087:
4088:         executors_file = self.repo_root / "src/farfan_core/core/orchestrator/executors.py"
4089:
4090:         if not executors_file.exists():
4091:             self.add_finding(
4092:                 AuditCategory.EXECUTOR_ARCHITECTURE,
4093:                 AuditStatus.FAILED,
4094:                 "executors.py",
4095:                 "Executors file not found",
4096:                 {"expected_path": str(executors_file)}
4097:             )
4098:             return {"status": "FAILED", "executors_found": 0}
4099:
4100:         # Parse the executors file
4101:         with open(executors_file, encoding='utf-8') as f:
4102:             content = f.read()
4103:
4104:         try:
4105:             tree = ast.parse(content)
4106:         except SyntaxError as e:
4107:             self.add_finding(
4108:                 AuditCategory.EXECUTOR_ARCHITECTURE,
4109:                 AuditStatus.FAILED,
4110:                 "executors.py",
4111:                 f"Syntax error in executors file: {e}",
4112:                 {"error": str(e)}
4113:             )
4114:             return {"status": "FAILED", "executors_found": 0}
4115:
```

```
        # Find all executor classes
        executor_classes = {}
        for node in ast.walk(tree):
            if isinstance(node, ast.ClassDef) and node.name.endswith("_Executor"):
                executor_classes[node.name] = {
                    "line_number": node.lineno,
                    "methods": [m.name for m in node.body if isinstance(m, ast.FunctionDef)]
                }

        # Audit each expected executor
        executor_audit_info = []
        found_count = 0

        for executor_name in self.EXPECTED_EXECUTORS:
            # Parse dimension and question from name (e.g., "D1Q2_Executor")
            parts = executor_name.replace("_Executor", "")
            dimension = int(parts[1])
            question = int(parts[3])
            dimension_name = self.DIMENSION_NAMES[dimension]

            class_exists = executor_name in executor_classes
            has_execute_method = False

            if class_exists:
                found_count += 1
                methods = executor_classes[executor_name]["methods"]
                has_execute_method = "execute" in methods

                status = AuditStatus.VERIFIED if has_execute_method else AuditStatus.WARNING
                message = "Executor properly defined" if has_execute_method else "Missing execute method"

                self.add_finding(
                    AuditCategory.EXECUTOR_ARCHITECTURE,
                    status,
                    executor_name,
                    message,
                    {
                        "dimension": f"D{dimension}: {dimension_name}",
                        "question": f"Q{question}",
                        "line": executor_classes[executor_name]["line_number"],
                        "methods": methods
                    }
                )
            else:
                self.add_finding(
                    AuditCategory.EXECUTOR_ARCHITECTURE,
                    AuditStatus.FAILED,
                    executor_name,
                    "Executor class not found",
                    {
                        "dimension": f"D{dimension}: {dimension_name}",
                        "question": f"Q{question}"
                    }
                )

            executor_audit_info.append(ExecutorAuditInfo(
```

```
                executor_name=executor_name,
                dimension=dimension,
                question=question,
                dimension_name=dimension_name,
                class_exists=class_exists,
                has_execute_method=has_execute_method,
                accesses_questionnaire_directly=False,  # Will be checked in questionnaire audit
                uses_dependency_injection=False,
                file_path=str(executors_file) if class_exists else None,
                line_number=executor_classes[executor_name]["line_number"] if class_exists else None
            ))

        # Overall assessment
        if found_count == 30:
            self.add_finding(
                AuditCategory.EXECUTOR_ARCHITECTURE,
                AuditStatus.VERIFIED,
                "FrontierExecutorOrchestrator",
                "All 30 dimension-question executors verified (D1Q1-D6Q5)",
                {
                    "expected": 30,
                    "found": found_count,
                    "dimensions": list(self.DIMENSION_NAMES.items())
                }
            )
        else:
            self.add_finding(
                AuditCategory.EXECUTOR_ARCHITECTURE,
                AuditStatus.FAILED,
                "FrontierExecutorOrchestrator",
                f"Missing executors: expected 30, found {found_count}",
                {
                    "expected": 30,
                    "found": found_count,
                    "missing": [e for e in self.EXPECTED_EXECUTORS if e not in executor_classes]
                }
            )

        return {
            "status": "VERIFIED" if found_count == 30 else "FAILED",
            "executors_found": found_count,
            "executors_expected": 30,
            "executor_details": executor_audit_info
        }

    def audit_questionnaire_access(self) -> dict[str, Any]:
        """
        Audit questionnaire access patterns to ensure dependency injection.

        Verifies that core scripts:
        1. Do NOT directly access questionnaire_monolith.json
        2. Do NOT instantiate QuestionnaireResourceProvider directly
        3. DO receive questionnaire via dependency injection

        Returns:
            Dictionary with audit results
```

```
        """
        logger.info("=" * 80)
        logger.info("AUDITING: Questionnaire Access Patterns (Dependency Injection)")
        logger.info("=" * 80)

        violations = []
        compliant_scripts = []

        for script_name in self.CORE_SCRIPTS:
            # Find the script in processing or analysis directories
            script_paths = [
                self.repo_root / "src/farfan_core/processing" / script_name,
                self.repo_root / "src/farfan_core/analysis" / script_name
            ]

            script_path = None
            for path in script_paths:
                if path.exists():
                    script_path = path
                    break

            if not script_path:
                self.add_finding(
                    AuditCategory.QUESTIONNAIRE_ACCESS,
                    AuditStatus.WARNING,
                    script_name,
                    "Script file not found",
                    {"searched_paths": [str(p) for p in script_paths]}
                )
                continue

            # Read and analyze the script
            with open(script_path, encoding='utf-8') as f:
                content = f.read()

            # Check for violations
            has_violations = False
            violation_details = []

            # Check for direct file access
            if 'questionnaire_monolith.json' in content or 'open(' in content and 'questionnaire' in content:
                has_violations = True
                violation_details.append("Direct file access to questionnaire detected")

            # Check for direct instantiation of QuestionnaireResourceProvider
            if 'QuestionnaireResourceProvider(' in content:
                has_violations = True
                violation_details.append("Direct instantiation of QuestionnaireResourceProvider")

            # Check for load_questionnaire() calls (should only be in factory)
            if 'load_questionnaire()' in content and script_name != 'factory.py':
                has_violations = True
                violation_details.append("Direct call to load_questionnaire()")

            # Verify dependency injection pattern
            uses_dependency_injection = False
```

```
            if (
                any(pattern in content for pattern in [
                    'questionnaire: Mapping',
                    'questionnaire: dict',
                ])
                or ('def __init__' in content and 'questionnaire' in content)
                or ('@dataclass' in content and 'questionnaire' in content)
            ):
                uses_dependency_injection = True

            if has_violations:
                violations.append(script_name)
                self.add_finding(
                    AuditCategory.QUESTIONNAIRE_ACCESS,
                    AuditStatus.FAILED,
                    script_name,
                    "Questionnaire access violations detected",
                    {
                        "violations": violation_details,
                        "file": str(script_path)
                    }
                )
            else:
                compliant_scripts.append(script_name)
                self.add_finding(
                    AuditCategory.QUESTIONNAIRE_ACCESS,
                    AuditStatus.VERIFIED,
                    script_name,
                    "Properly uses dependency injection for questionnaire access",
                    {
                        "uses_dependency_injection": uses_dependency_injection,
                        "file": str(script_path)
                    }
                )

        # Check factory.py as the authorized loader
        factory_path = self.repo_root / "src/farfan_core/core/orchestrator/factory.py"
        if factory_path.exists():
            with open(factory_path, encoding='utf-8') as f:
                factory_content = f.read()

            has_load_function = 'load_questionnaire' in factory_content
            has_provider_creation = 'QuestionnaireResourceProvider' in factory_content

            if has_load_function:
                self.add_finding(
                    AuditCategory.QUESTIONNAIRE_ACCESS,
                    AuditStatus.VERIFIED,
                    "factory.py",
                    "Authorized questionnaire loader verified",
                    {
                        "has_load_function": has_load_function,
                        "has_provider_creation": has_provider_creation
                    }
                )
            else:
```

```
                self.add_finding(
                    AuditCategory.QUESTIONNAIRE_ACCESS,
                    AuditStatus.WARNING,
                    "factory.py",
                    "Factory missing questionnaire loading functionality"
                )

        # Overall assessment
        total_scripts = len(self.CORE_SCRIPTS)
        compliant_count = len(compliant_scripts)

        if compliant_count == total_scripts:
            self.add_finding(
                AuditCategory.QUESTIONNAIRE_ACCESS,
                AuditStatus.VERIFIED,
                "Questionnaire Access Policy",
                f"All {total_scripts} core scripts comply with dependency injection policy",
                {
                    "compliant_scripts": compliant_scripts,
                    "violations": []
                }
            )
        else:
            self.add_finding(
                AuditCategory.QUESTIONNAIRE_ACCESS,
                AuditStatus.FAILED,
                "Questionnaire Access Policy",
                f"Policy violations found: {len(violations)}/{total_scripts} scripts",
                {
                    "compliant_scripts": compliant_scripts,
                    "violations": violations
                }
            )

        return {
            "status": "VERIFIED" if compliant_count == total_scripts else "FAILED",
            "compliant_scripts": compliant_count,
            "total_scripts": total_scripts,
            "violations": violations
        }

    def audit_factory_pattern(self) -> dict[str, Any]:
        """
        Audit factory pattern implementation.

        Verifies:
        1. Primary loader exists: factory.py::load_questionnaire_monolith()
        2. QuestionnaireResourceProvider for dependency injection
        3. No unauthorized direct access

        Returns:
            Dictionary with audit results
        """
        logger.info("=" * 80)
        logger.info("AUDITING: Factory Pattern Implementation")
        logger.info("=" * 80)
```

```
4396:
4397:         factory_path = self.repo_root / "src/farfan_core/core/orchestrator/factory.py"
4398:         questionnaire_path = self.repo_root / "src/farfan_core/core/orchestrator/questionnaire.py"
4399:
4400:         results = {
4401:             "factory_exists": False,
4402:             "has_load_function": False,
4403:             "questionnaire_module_exists": False,
4404:             "has_provider_class": False
4405:         }
4406:
4407:         # Check factory.py
4408:         if factory_path.exists():
4409:             results["factory_exists"] = True
4410:             with open(factory_path, encoding='utf-8') as f:
4411:                 factory_content = f.read()
4412:
4413:             # Parse AST
4414:             try:
4415:                 tree = ast.parse(factory_content)
4416:
4417:                 # Look for load functions
4418:                 for node in ast.walk(tree):
4419:                     if isinstance(node, ast.FunctionDef):
4420:                         if 'load_questionnaire' in node.name.lower():
4421:                             results["has_load_function"] = True
4422:                             self.add_finding(
4423:                                 AuditCategory.FACTORY_PATTERN,
4424:                                 AuditStatus.VERIFIED,
4425:                                 f"factory.py::{node.name}",
4426:                                 "Questionnaire loader function found",
4427:                                 {"line": node.lineno}
4428:                             )
4429:             except SyntaxError as e:
4430:                 self.add_finding(
4431:                     AuditCategory.FACTORY_PATTERN,
4432:                     AuditStatus.FAILED,
4433:                     "factory.py",
4434:                     f"Syntax error: {e}"
4435:                 )
4436:         else:
4437:             self.add_finding(
4438:                 AuditCategory.FACTORY_PATTERN,
4439:                 AuditStatus.FAILED,
4440:                 "factory.py",
4441:                 "Factory file not found",
4442:                 {"expected_path": str(factory_path)}
4443:             )
4444:
4445:         # Check questionnaire.py
4446:         if questionnaire_path.exists():
4447:             results["questionnaire_module_exists"] = True
4448:             with open(questionnaire_path, encoding='utf-8') as f:
4449:                 questionnaire_content = f.read()
4450:
4451:             # Parse AST
```

```
            try:
                tree = ast.parse(questionnaire_content)

                # Look for QuestionnaireResourceProvider
                for node in ast.walk(tree):
                    if isinstance(node, ast.ClassDef):
                        if 'QuestionnaireResourceProvider' in node.name:
                            results["has_provider_class"] = True
                            self.add_finding(
                                AuditCategory.FACTORY_PATTERN,
                                AuditStatus.VERIFIED,
                                f"questionnaire.py::{node.name}",
                                "QuestionnaireResourceProvider class found",
                                {"line": node.lineno}
                            )
            except SyntaxError as e:
                self.add_finding(
                    AuditCategory.FACTORY_PATTERN,
                    AuditStatus.FAILED,
                    "questionnaire.py",
                    f"Syntax error: {e}"
                )
        else:
            self.add_finding(
                AuditCategory.FACTORY_PATTERN,
                AuditStatus.FAILED,
                "questionnaire.py",
                "Questionnaire module not found",
                {"expected_path": str(questionnaire_path)}
            )

        # Overall assessment
        all_verified = all(results.values())

        if all_verified:
            self.add_finding(
                AuditCategory.FACTORY_PATTERN,
                AuditStatus.VERIFIED,
                "Factory Pattern",
                "Factory pattern fully implemented and verified",
                results
            )
        else:
            self.add_finding(
                AuditCategory.FACTORY_PATTERN,
                AuditStatus.FAILED,
                "Factory Pattern",
                "Factory pattern incomplete or missing components",
                results
            )

        return {
            "status": "VERIFIED" if all_verified else "FAILED",
            **results
        }

```

```
    def audit_method_signatures(self) -> dict[str, Any]:
        """
        Audit method signatures across core modules.

        Verifies that all methods have:
        1. Type annotations
        2. Docstrings
        3. Proper parameter documentation

        Returns:
            Dictionary with audit results
        """
        logger.info("=" * 80)
        logger.info("AUDITING: Method Signatures (165 methods across 38 classes)")
        logger.info("=" * 80)

        # Target files to audit
        target_files = [
            self.repo_root / "src/farfan_core/processing" / script
            for script in self.CORE_SCRIPTS
        ] + [
            self.repo_root / "src/farfan_core/analysis" / script
            for script in self.CORE_SCRIPTS
        ]

        total_methods = 0
        complete_methods = 0
        incomplete_methods = []

        for file_path in target_files:
            if not file_path.exists():
                continue

            with open(file_path, encoding='utf-8') as f:
                content = f.read()

            try:
                tree = ast.parse(content)

                for node in ast.walk(tree):
                    if isinstance(node, ast.ClassDef):
                        class_name = node.name

                        for item in node.body:
                            if isinstance(item, ast.FunctionDef):
                                total_methods += 1
                                method_name = item.name

                                # Check for type annotations
                                has_return_annotation = item.returns is not None
                                has_param_annotations = all(
                                    arg.annotation is not None
                                    for arg in item.args.args
                                    if arg.arg != 'self'
                                )

```

```
                                # Check for docstring
                                has_docstring = (
                                    ast.get_docstring(item) is not None
                                )

                                is_complete = (
                                    has_return_annotation and
                                    has_param_annotations and
                                    has_docstring
                                )

                                if is_complete:
                                    complete_methods += 1
                                else:
                                    incomplete_methods.append({
                                        "file": file_path.name,
                                        "class": class_name,
                                        "method": method_name,
                                        "line": item.lineno,
                                        "missing": {
                                            "return_annotation": not has_return_annotation,
                                            "param_annotations": not has_param_annotations,
                                            "docstring": not has_docstring
                                        }
                                    })
            except SyntaxError as e:
                logger.warning(f"Syntax error in {file_path}: {e}")

        # Assessment
        completion_rate = (complete_methods / total_methods * 100) if total_methods > 0 else 0

        if completion_rate == 100:
            self.add_finding(
                AuditCategory.METHOD_SIGNATURES,
                AuditStatus.VERIFIED,
                "Method Signatures",
                f"All {total_methods} methods have complete signatures",
                {
                    "total": total_methods,
                    "complete": complete_methods,
                    "completion_rate": completion_rate
                }
            )
        elif completion_rate >= 90:
            self.add_finding(
                AuditCategory.METHOD_SIGNATURES,
                AuditStatus.WARNING,
                "Method Signatures",
                f"Most methods complete ({completion_rate:.1f}%), but {len(incomplete_methods)} need attention",
                {
                    "total": total_methods,
                    "complete": complete_methods,
                    "incomplete": len(incomplete_methods),
                    "completion_rate": completion_rate
                }
            )
```

```
4620:         else:
4621:             self.add_finding(
4622:                 AuditCategory.METHOD_SIGNATURES,
4623:                 AuditStatus.FAILED,
4624:                 "Method Signatures",
4625:                 f"Insufficient completion rate ({completion_rate:.1f}%)",
4626:                 {
4627:                     "total": total_methods,
4628:                     "complete": complete_methods,
4629:                     "incomplete": len(incomplete_methods),
4630:                     "completion_rate": completion_rate,
4631:                     "incomplete_methods": incomplete_methods[:10]  # First 10
4632:                 }
4633:             )
4634:
4635:         return {
4636:             "status": "VERIFIED" if completion_rate == 100 else ("WARNING" if completion_rate >= 90 else "FAILED"),
4637:             "total_methods": total_methods,
4638:             "complete_methods": complete_methods,
4639:             "completion_rate": completion_rate,
4640:             "incomplete_methods": incomplete_methods
4641:         }
4642:
4643:     def audit_configuration_system(self) -> dict[str, Any]:
4644:         """
4645:         Audit configuration system for type-safety and parameters.
4646:
4647:         Verifies:
4648:         1. ExecutorConfig with proper parameter ranges
4649:         2. AdvancedModuleConfig with academic parameters
4650:         3. Type-safety with Pydantic
4651:         4. BLAKE3 fingerprinting
4652:
4653:         Returns:
4654:             Dictionary with audit results
4655:         """
4656:         logger.info("=" * 80)
4657:         logger.info("AUDITING: Configuration System (Type-Safety & Parameters)")
4658:         logger.info("=" * 80)
4659:
4660:         config_files = {
4661:             "executor_config": self.repo_root / "src/farfan_core/core/orchestrator/executor_config.py",
4662:             "advanced_module_config": self.repo_root / "src/farfan_core/core/orchestrator/advanced_module_config.py"
4663:         }
4664:
4665:         results = {}
4666:
4667:         for config_name, config_path in config_files.items():
4668:             if not config_path.exists():
4669:                 self.add_finding(
4670:                     AuditCategory.CONFIGURATION_SYSTEM,
4671:                     AuditStatus.FAILED,
4672:                     config_name,
4673:                     "Configuration file not found",
4674:                     {"expected_path": str(config_path)}
4675:                 )
```

```
                results[config_name] = False
                continue

            with open(config_path, encoding='utf-8') as f:
                content = f.read()

            # Check for Pydantic BaseModel
            has_pydantic = 'BaseModel' in content or 'pydantic' in content
            has_field_validation = 'Field(' in content or 'validator' in content
            has_frozen = 'frozen=True' in content or 'class Config' in content

            if has_pydantic and has_field_validation:
                self.add_finding(
                    AuditCategory.CONFIGURATION_SYSTEM,
                    AuditStatus.VERIFIED,
                    config_name,
                    "Type-safe configuration with Pydantic validation",
                    {
                        "has_pydantic": has_pydantic,
                        "has_field_validation": has_field_validation,
                        "has_frozen": has_frozen
                    }
                )
                results[config_name] = True
            else:
                self.add_finding(
                    AuditCategory.CONFIGURATION_SYSTEM,
                    AuditStatus.WARNING,
                    config_name,
                    "Configuration lacks proper type-safety features",
                    {
                        "has_pydantic": has_pydantic,
                        "has_field_validation": has_field_validation,
                        "has_frozen": has_frozen
                    }
                )
                results[config_name] = False

        # Overall assessment
        all_verified = all(results.values())

        if all_verified:
            self.add_finding(
                AuditCategory.CONFIGURATION_SYSTEM,
                AuditStatus.VERIFIED,
                "Configuration System",
                "All configuration modules are type-safe and properly validated",
                results
            )
        else:
            self.add_finding(
                AuditCategory.CONFIGURATION_SYSTEM,
                AuditStatus.WARNING,
                "Configuration System",
                "Some configuration modules need improvement",
                results
```

```
            )

        return {
            "status": "VERIFIED" if all_verified else "WARNING",
            **results
        }

    def generate_audit_report(self, output_path: Path | None = None) -> dict[str, Any]:
        """
        Generate comprehensive audit report.

        Args:
            output_path: Optional path to save the report

        Returns:
            Complete audit report as dictionary
        """
        logger.info("=" * 80)
        logger.info("GENERATING COMPREHENSIVE AUDIT REPORT")
        logger.info("=" * 80)

        # Run all audits
        executor_results = self.audit_executor_architecture()
        questionnaire_results = self.audit_questionnaire_access()
        factory_results = self.audit_factory_pattern()
        method_results = self.audit_method_signatures()
        config_results = self.audit_configuration_system()

        # Compile report
        report = {
            "audit_metadata": {
                "timestamp": datetime.utcnow().isoformat(),
                "repository_root": str(self.repo_root),
                "total_findings": len(self.findings)
            },
            "audit_results": {
                "executor_architecture": executor_results,
                "questionnaire_access": questionnaire_results,
                "factory_pattern": factory_results,
                "method_signatures": method_results,
                "configuration_system": config_results
            },
            "findings": [f.to_dict() for f in self.findings],
            "summary": self._generate_summary()
        }

        # Save report if path provided
        if output_path:
            output_path.parent.mkdir(parents=True, exist_ok=True)
            with open(output_path, 'w', encoding='utf-8') as f:
                json.dump(report, f, indent=2)
            logger.info(f"Audit report saved to: {output_path}")

        return report

    def _generate_summary(self) -> dict[str, Any]:
```

```
4788:         """Generate summary of audit findings."""
4789:         summary = {
4790:             "total_findings": len(self.findings),
4791:             "verified": sum(1 for f in self.findings if f.status == AuditStatus.VERIFIED),
4792:             "warnings": sum(1 for f in self.findings if f.status == AuditStatus.WARNING),
4793:             "failed": sum(1 for f in self.findings if f.status == AuditStatus.FAILED),
4794:             "by_category": {}
4795:         }
4796:
4797:         # Group by category
4798:         for category in AuditCategory:
4799:             category_findings = [f for f in self.findings if f.category == category]
4800:             summary["by_category"][category.value] = {
4801:                 "total": len(category_findings),
4802:                 "verified": sum(1 for f in category_findings if f.status == AuditStatus.VERIFIED),
4803:                 "warnings": sum(1 for f in category_findings if f.status == AuditStatus.WARNING),
4804:                 "failed": sum(1 for f in category_findings if f.status == AuditStatus.FAILED)
4805:             }
4806:
4807:         return summary
4808:
4809:     def print_summary(self) -> None:
4810:         """Print audit summary to console."""
4811:         summary = self._generate_summary()
4812:
4813:         print("\n" + "=" * 80)
4814:         print("ð\237\223\213 AUDIT SUMMARY")
4815:         print("=" * 80)
4816:         print(f"Total Findings: {summary['total_findings']}")
4817:         print(f"  â\234\205 Verified: {summary['verified']}")
4818:         print(f"  â\232 ï¸\217  Warnings: {summary['warnings']}")
4819:         print(f"  â\235\214 Failed: {summary['failed']}")
4820:         print("\n" + "-" * 80)
4821:         print("By Category:")
4822:         print("-" * 80)
4823:
4824:         for category, stats in summary["by_category"].items():
4825:             if stats["total"] > 0:
4826:                 print(f"\n{category}:")
4827:                 print(f"  Total: {stats['total']}")
4828:                 print(f"  â\234\205 Verified: {stats['verified']}")
4829:                 print(f"  â\232 ï¸\217  Warnings: {stats['warnings']}")
4830:                 print(f"  â\235\214 Failed: {stats['failed']}")
4831:
4832:         print("\n" + "=" * 80)
4833:
4834:
4835: def main() -> None:
4836:     """Main entry point for audit system."""
4837:     import argparse
4838:
4839:     parser = argparse.ArgumentParser(description="FARFAN Pipeline Audit System")
4840:     parser.add_argument(
4841:         "--repo-root",
4842:         type=Path,
4843:         default=PROJECT_ROOT,
```

```
        help="Repository root directory"
    )
    parser.add_argument(
        "--output",
        type=Path,
        help="Output path for audit report (JSON)"
    )
    parser.add_argument(
        "--verbose",
        action="store_true",
        help="Enable verbose logging"
    )

    args = parser.parse_args()

    # Configure logging
    logging.basicConfig(
        level=logging.DEBUG if args.verbose else logging.INFO,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    )

    # Run audit
    audit_system = AuditSystem(args.repo_root)
    report = audit_system.generate_audit_report(args.output)
    audit_system.print_summary()

    # Exit with appropriate code
    if report["summary"]["failed"] > 0:
        sys.exit(1)
    elif report["summary"]["warnings"] > 0:
        sys.exit(2)
    else:
        sys.exit(0)


if __name__ == "__main__":
    main()



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

"""
Compatibility Layer - Version Shims and Polyfills

This module provides a unified interface for Python version differences
and third-party package variations. All version-specific imports should
go through this layer.

Shims provided:
- tomllib/tomli (TOML parsing, Python 3.11+ vs earlier)
- importlib.resources (files() API, Python 3.9+ vs earlier)
- typing extensions (backports for older Python)
- typing (future annotations support)
```

```
4900:
4901: Design:
4902: - Explicit imports only (no star imports)
4903: - Fail-fast on missing required compatibility
4904: - Clear error messages with version requirements
4905: """
4906:
4907: from __future__ import annotations
4908:
4909: import sys
4910: from typing import Any
4911:
4912: # Lazy loading utilities for heavy dependencies
4913: from farfan_pipeline.compat.lazy_deps import (
4914:     get_numpy,
4915:     get_pandas,
4916:     get_polars,
4917:     get_pyarrow,
4918:     get_spacy,
4919:     get_torch,
4920:     get_transformers,
4921: )
4922: from farfan_pipeline.compat.safe_imports import (
4923:     ImportErrorDetailed,
4924:     check_import_available,
4925:     get_import_version,
4926:     lazy_import,
4927:     try_import,
4928: )
4929:
4930: # Backward compatibility alias
4931: OptionalDependencyError = ImportErrorDetailed
4932:
4933: # Re-export safe import utilities and lazy deps
4934: __all__ = [
4935:     # Core import utilities
4936:     "ImportErrorDetailed",
4937:     "OptionalDependencyError",  # Backward compatibility alias
4938:     "try_import",
4939:     "lazy_import",
4940:     "check_import_available",
4941:     "get_import_version",
4942:     # Version compatibility shims
4943:     "tomllib",
4944:     "resources_files",
4945:     # Lazy loading utilities
4946:     "get_numpy",
4947:     "get_pandas",
4948:     "get_polars",
4949:     "get_pyarrow",
4950:     "get_torch",
4951:     "get_transformers",
4952:     "get_spacy",
4953: ]
4954:
4955:
```

```
# ============================================================================
# TOML Parsing - Python 3.11+ tomllib vs tomli
# ============================================================================

if sys.version_info >= (3, 11):
    import tomllib
else:
    # Python < 3.11 needs tomli package
    # try_import with required=True will raise if missing
    tomllib = try_import(  # type: ignore[assignment]
        "tomli",
        required=True,
        hint="Python < 3.11 requires 'tomli' package. Install with: pip install tomli",
    )


# ============================================================================
# Importlib Resources - Python 3.9+ files() vs older resource API
# ============================================================================

try:
    from importlib.resources import files as resources_files
except ImportError:
    # Python < 3.9 needs importlib_resources backport
    # try_import with required=True will raise if missing
    _resources = try_import(
        "importlib_resources",
        required=True,
        hint="Python < 3.9 requires 'importlib_resources'. "
             "Install with: pip install importlib-resources",
    )
    resources_files = _resources.files  # type: ignore[attr-defined]


# ============================================================================
# Typing Extensions - Backports for older Python versions
# ============================================================================

# For maximum compatibility, we always try to import typing_extensions
# Even on Python 3.10+, typing_extensions provides latest features
_typing_extensions_available = check_import_available("typing_extensions")

if _typing_extensions_available:
    import typing_extensions

    # Use typing_extensions versions if available (they're usually more up-to-date)
    TypeAlias = typing_extensions.TypeAlias
    ParamSpec = typing_extensions.ParamSpec
    Concatenate = typing_extensions.Concatenate
    Literal = typing_extensions.Literal
    Protocol = typing_extensions.Protocol
    TypedDict = typing_extensions.TypedDict
    Final = typing_extensions.Final
    Annotated = typing_extensions.Annotated

else:
```

```
    # Fall back to stdlib typing
    # This may not have all features on older Python versions
    # TypeAlias added in 3.10
    from typing import (  # type: ignore[misc, assignment]
        Annotated,  # 3.9+
        Concatenate,
        Final,  # 3.8+
        Literal,  # 3.8+
        ParamSpec,
        Protocol,  # 3.8+
        TypeAlias,
        TypedDict,  # 3.8+
    )


# ============================================================================
# Platform Detection Utilities
# ============================================================================

def get_platform_info() -> dict[str, Any]:
    """
    Get comprehensive platform information for debugging.

    Returns
    -------
    dict[str, Any]
        Platform details including OS, architecture, Python version

    Examples
    --------
    >>> info = get_platform_info()
    >>> print(f"Running on {info['system']} {info['architecture']}")
    """
    import platform

    return {
        "system": platform.system(),
        "release": platform.release(),
        "version": platform.version(),
        "architecture": platform.machine(),
        "python_version": sys.version,
        "python_implementation": platform.python_implementation(),
        "python_version_tuple": sys.version_info[:3],
    }


def check_minimum_python_version(major: int, minor: int) -> bool:
    """
    Check if Python version meets minimum requirement.

    Parameters
    ----------
    major : int
        Required major version
    minor : int
        Required minor version
```

```
5068:
5069:     Returns
5070:     -------
5071:     bool
5072:         True if current version >= required version
5073:
5074:     Examples
5075:     --------
5076:     >>> if not check_minimum_python_version(3, 10):
5077:     ...     raise RuntimeError("Python 3.10+ required")
5078:     """
5079:     return sys.version_info >= (major, minor)
5080:
5081:
5082: # ============================================================================
5083: # Validation on Import
5084: # ============================================================================
5085:
5086: # Ensure minimum Python version (as specified in pyproject.toml)
5087: if not check_minimum_python_version(3, 10):
5088:     raise ImportErrorDetailed(
5089:         f"Python 3.10 or later required. Current version: {sys.version}. "
5090:         "Please upgrade Python or use a compatible environment."
5091:     )
5092:
5093:
5094:
5095: ================================================================================
5096: FILE: src/farfan_pipeline/compat/lazy_deps.py
5097: ================================================================================
5098:
5099: """
5100: Lazy Loading for Heavy Dependencies
5101:
5102: This module provides lazy-loaded imports for heavy optional dependencies
5103: to reduce import-time overhead and improve startup performance.
5104:
5105: Heavy dependencies include:
5106: - polars: Fast DataFrame library (50-200ms import time)
5107: - pyarrow: Arrow format support (50-150ms import time)
5108: - torch: Deep learning framework (500-1500ms import time)
5109: - tensorflow: Machine learning framework (1000-3000ms import time)
5110: - transformers: NLP models (200-500ms import time)
5111: - spacy: NLP processing (200-400ms import time)
5112:
5113: Usage:
5114:     from farfan_core.compat.lazy_deps import get_polars, get_pyarrow
5115:
5116:     def process_dataframe(data):
5117:         pl = get_polars()  # Lazy-loaded on first call
5118:         return pl.DataFrame(data)
5119: """
5120:
5121: from __future__ import annotations
5122:
5123: from typing import Any
```

```
5124:
5125: from farfan_pipeline.compat.safe_imports import lazy_import
5126:
5127:
5128: def get_polars() -> Any:
5129:     """
5130:     Lazy-load polars library.
5131:
5132:     Returns
5133:     -------
5134:     module
5135:         The polars module
5136:
5137:     Raises
5138:     ------
5139:     ImportErrorDetailed
5140:         If polars is not installed
5141:
5142:     Examples
5143:     --------
5144:     >>> pl = get_polars()
5145:     >>> df = pl.DataFrame({"a": [1, 2, 3]})
5146:     """
5147:     return lazy_import(
5148:         "polars",
5149:         hint="Install with: pip install polars\n"
5150:              "Or install analytics extra: pip install farfan_core[analytics]"
5151:     )
5152:
5153:
5154: def get_pyarrow() -> Any:
5155:     """
5156:     Lazy-load pyarrow library.
5157:
5158:     Returns
5159:     -------
5160:     module
5161:         The pyarrow module
5162:
5163:     Raises
5164:     ------
5165:     ImportErrorDetailed
5166:         If pyarrow is not installed
5167:
5168:     Examples
5169:     --------
5170:     >>> pa = get_pyarrow()
5171:     >>> table = pa.table({"a": [1, 2, 3]})
5172:     """
5173:     return lazy_import(
5174:         "pyarrow",
5175:         hint="Install with: pip install pyarrow\n"
5176:              "This is a core dependency for Arrow format support."
5177:     )
5178:
5179:
```

```
def get_torch() -> Any:
    """
    Lazy-load torch library.

    Returns
    -------
    module
        The torch module

    Raises
    ------
    ImportErrorDetailed
        If torch is not installed

    Examples
    --------
    >>> torch = get_torch()
    >>> tensor = torch.tensor([1, 2, 3])
    """
    return lazy_import(
        "torch",
        hint="Install with: pip install torch\n"
             "Or install ml extra: pip install farfan_core[ml]"
    )


def get_tensorflow() -> Any:
    """
    Lazy-load tensorflow library.

    Returns
    -------
    module
        The tensorflow module

    Raises
    ------
    ImportErrorDetailed
        If tensorflow is not installed

    Examples
    --------
    >>> tf = get_tensorflow()
    >>> tensor = tf.constant([1, 2, 3])
    """
    return lazy_import(
        "tensorflow",
        hint="Install with: pip install tensorflow\n"
             "Or install ml extra: pip install farfan_core[ml]"
    )


def get_transformers() -> Any:
    """
    Lazy-load transformers library.

```

```
    Returns
    -------
    module
        The transformers module

    Raises
    ------
    ImportErrorDetailed
        If transformers is not installed

    Examples
    --------
    >>> transformers = get_transformers()
    >>> model = transformers.AutoModel.from_pretrained("bert-base-uncased")
    """
    return lazy_import(
        "transformers",
        hint="Install with: pip install transformers\n"
             "Or install nlp extra: pip install farfan_core[nlp]"
    )


def get_spacy() -> Any:
    """
    Lazy-load spacy library.

    Returns
    -------
    module
        The spacy module

    Raises
    ------
    ImportErrorDetailed
        If spacy is not installed

    Examples
    --------
    >>> spacy = get_spacy()
    >>> nlp = spacy.load("es_core_news_sm")
    """
    return lazy_import(
        "spacy",
        hint="Install with: pip install spacy\n"
             "Then download language model: python -m spacy download es_core_news_sm\n"
             "Or install nlp extra: pip install farfan_core[nlp]"
    )


def get_pandas() -> Any:
    """
    Lazy-load pandas library.

    This is typically a required dependency but we lazy-load it
    to reduce import-time overhead.

```

```
    Returns
    -------
    module
        The pandas module

    Raises
    ------
    ImportErrorDetailed
        If pandas is not installed

    Examples
    --------
    >>> pd = get_pandas()
    >>> df = pd.DataFrame({"a": [1, 2, 3]})
    """
    return lazy_import(
        "pandas",
        hint="Install with: pip install pandas\n"
             "This is a core dependency."
    )


def get_numpy() -> Any:
    """
    Lazy-load numpy library.

    This is typically a required dependency but we lazy-load it
    to reduce import-time overhead in modules that don't always need it.

    Returns
    -------
    module
        The numpy module

    Raises
    ------
    ImportErrorDetailed
        If numpy is not installed

    Examples
    --------
    >>> np = get_numpy()
    >>> array = np.array([1, 2, 3])
    """
    return lazy_import(
        "numpy",
        hint="Install with: pip install numpy\n"
             "This is a core dependency."
    )


# Convenience mapping for programmatic access
LAZY_DEPS = {
    "polars": get_polars,
    "pyarrow": get_pyarrow,
    "torch": get_torch,
```

```
    "tensorflow": get_tensorflow,
    "transformers": get_transformers,
    "spacy": get_spacy,
    "pandas": get_pandas,
    "numpy": get_numpy,
}


def get_lazy_dep(name: str) -> Any:
    """
    Get a lazy-loaded dependency by name.

    Parameters
    ----------
    name : str
        Name of the dependency (e.g., "polars", "torch")

    Returns
    -------
    module
        The requested module

    Raises
    ------
    KeyError
        If the dependency name is not recognized
    ImportErrorDetailed
        If the dependency is not installed

    Examples
    --------
    >>> polars = get_lazy_dep("polars")
    >>> torch = get_lazy_dep("torch")
    """
    if name not in LAZY_DEPS:
        raise KeyError(
            f"Unknown lazy dependency: {name}. "
            f"Available: {', '.join(LAZY_DEPS.keys())}"
        )

    return LAZY_DEPS[name]()


__all__ = [
    "get_polars",
    "get_pyarrow",
    "get_torch",
    "get_tensorflow",
    "get_transformers",
    "get_spacy",
    "get_pandas",
    "get_numpy",
    "get_lazy_dep",
    "LAZY_DEPS",
]

```

```
5404:
5405:
5406: ================================================================================
5407: FILE: src/farfan_pipeline/compat/native_check.py
5408: ================================================================================
5409:
5410: """
5411: Native Extension and System Library Verification
5412:
5413: This module checks for C-extensions, native libraries, and platform-specific
5414: requirements to provide early detection and clear error messages.
5415:
5416: Checks include:
5417: - Wheel compatibility (manylinux, musllinux, macosx, win)
5418: - System libraries (libzstd, icu, libomp, libstdc++)
5419: - CPU features (AVX, NEON for polars/arrow)
5420: - FIPS mode detection for cryptography
5421: """
5422:
5423: from __future__ import annotations
5424:
5425: import os
5426: import platform
5427: import subprocess
5428: import sys
5429: from dataclasses import dataclass
5430: from pathlib import Path
5431:
5432:
5433: @dataclass
5434: class NativeCheckResult:
5435:     """Result of a native library or extension check."""
5436:
5437:     available: bool
5438:     message: str
5439:     hint: str = ""
5440:
5441:
5442: def check_system_library(libname: str) -> NativeCheckResult:
5443:     """
5444:     Check if a system library is available.
5445:
5446:     Parameters
5447:     ----------
5448:     libname : str
5449:         Library name (e.g., 'zstd', 'icu', 'omp')
5450:
5451:     Returns
5452:     -------
5453:     NativeCheckResult
5454:         Result with availability and guidance
5455:
5456:     Examples
5457:     --------
5458:     >>> result = check_system_library('zstd')
5459:     >>> if not result.available:
```

```
    ...     print(result.hint)
    """
    system = platform.system()

    if system == "Linux":
        # Try ldconfig or direct file check
        try:
            result = subprocess.run(
                ["ldconfig", "-p"],
                check=False, capture_output=True,
                text=True,
                timeout=5,
            )
            if libname in result.stdout:
                return NativeCheckResult(
                    available=True,
                    message=f"Library {libname} found via ldconfig",
                )
        except (subprocess.SubprocessError, FileNotFoundError):
            pass

        # Fallback: check common lib paths
        base_root = Path(os.sep)
        common_paths = [
            base_root / "usr" / "lib" / f"lib{libname}.so",
            base_root / "usr" / "lib" / "x86_64-linux-gnu" / f"lib{libname}.so",
            base_root / "usr" / "local" / "lib" / f"lib{libname}.so",
        ]
        for path in common_paths:
            if path.exists():
                return NativeCheckResult(
                    available=True,
                    message=f"Library {libname} found at {path}",
                )

        return NativeCheckResult(
            available=False,
            message=f"Library {libname} not found",
            hint=f"Install system package: apt-get install lib{libname}-dev (Debian/Ubuntu) "
                 f"or yum install {libname}-devel (RHEL/CentOS)",
        )

    elif system == "Darwin":
        # macOS - check via dyld
        try:
            result = subprocess.run(
                ["otool", "-L", sys.executable],
                check=False, capture_output=True,
                text=True,
                timeout=5,
            )
            if libname in result.stdout:
                return NativeCheckResult(
                    available=True,
                    message=f"Library {libname} found via otool",
                )
```

```
5516:         except (subprocess.SubprocessError, FileNotFoundError):
5517:             pass
5518:
5519:         return NativeCheckResult(
5520:             available=False,
5521:             message=f"Library {libname} not found",
5522:             hint=f"Install via Homebrew: brew install {libname}",
5523:         )
5524:
5525:     elif system == "Windows":
5526:         # Windows - check PATH and common locations
5527:         for path_dir in os.environ.get("PATH", "").split(os.pathsep):
5528:             dll_path = Path(path_dir) / f"{libname}.dll"
5529:             if dll_path.exists():
5530:                 return NativeCheckResult(
5531:                     available=True,
5532:                     message=f"Library {libname}.dll found at {dll_path}",
5533:                 )
5534:
5535:         return NativeCheckResult(
5536:             available=False,
5537:             message=f"Library {libname}.dll not found in PATH",
5538:             hint=f"Install {libname} and add to PATH",
5539:         )
5540:
5541:     return NativeCheckResult(
5542:         available=False,
5543:         message=f"Cannot check library {libname} on {system}",
5544:         hint="Platform detection not implemented for this system",
5545:     )
5546:
5547:
5548: def check_wheel_compatibility(package: str) -> NativeCheckResult:
5549:     """
5550:     Check if a package has appropriate wheels for the current platform.
5551:
5552:     Parameters
5553:     ----------
5554:     package : str
5555:         Package name (e.g., 'pyarrow', 'polars')
5556:
5557:     Returns
5558:     -------
5559:     NativeCheckResult
5560:         Compatibility status and guidance
5561:     """
5562:     system = platform.system()
5563:     platform.machine()
5564:
5565:     # Platform tags we expect
5566:     if system in {"Linux", "Darwin", "Windows"}:
5567:         pass
5568:
5569:     try:
5570:         # Try to import and check __file__ for wheel origin
5571:         import importlib
```

```
        mod = importlib.import_module(package)
        if hasattr(mod, "__file__") and mod.__file__:
            file_path = mod.__file__
            # Check if installed from wheel (dist-info present)
            if "site-packages" in file_path:
                return NativeCheckResult(
                    available=True,
                    message=f"Package {package} installed from wheel",
                )

        return NativeCheckResult(
            available=True,
            message=f"Package {package} available (source install or wheel)",
            hint="Consider using pre-built wheels for better compatibility",
        )
    except ImportError:
        return NativeCheckResult(
            available=False,
            message=f"Package {package} not installed",
            hint=f"Install with: pip install {package}",
        )


def check_cpu_features() -> NativeCheckResult:
    """
    Check CPU features required by performance libraries.

    Some packages (polars, pyarrow) require specific CPU instructions.

    Returns
    -------
    NativeCheckResult
        CPU feature availability
    """
    machine = platform.machine().lower()

    # Basic architecture check
    if machine in ("x86_64", "amd64"):
        # Would need cpuinfo or similar for detailed AVX detection
        # For now, assume modern x86_64 has basic features
        return NativeCheckResult(
            available=True,
            message=f"CPU architecture {machine} is supported",
        )
    elif machine in ("arm64", "aarch64"):
        return NativeCheckResult(
            available=True,
            message=f"CPU architecture {machine} (ARM) is supported",
        )
    else:
        return NativeCheckResult(
            available=False,
            message=f"CPU architecture {machine} may not be supported",
            hint="Some packages require x86_64 or ARM64. Check package documentation.",
        )

```

```
5628:
5629: def check_fips_mode() -> bool:
5630:     """
5631:     Detect if system is in FIPS mode (Federal Information Processing Standards).
5632:
5633:     This affects cryptography backend selection.
5634:
5635:     Returns
5636:     -------
5637:     bool
5638:         True if FIPS mode is enabled
5639:     """
5640:     if platform.system() == "Linux":
5641:         # Check /proc/sys/crypto/fips_enabled
5642:         try:
5643:             with open("/proc/sys/crypto/fips_enabled") as f:
5644:                 return f.read().strip() == "1"
5645:         except (FileNotFoundError, PermissionError):
5646:             pass
5647:
5648:     return False
5649:
5650:
5651: def verify_native_dependencies(packages: list[str]) -> dict[str, NativeCheckResult]:
5652:     """
5653:     Verify native dependencies for a list of packages.
5654:
5655:     Parameters
5656:     ----------
5657:     packages : list[str]
5658:         Package names to verify
5659:
5660:     Returns
5661:     -------
5662:     dict[str, NativeCheckResult]
5663:         Mapping of package name to verification result
5664:
5665:     Examples
5666:     --------
5667:     >>> results = verify_native_dependencies(['pyarrow', 'polars'])
5668:     >>> for pkg, result in results.items():
5669:     ...     if not result.available:
5670:     ...         print(f"{pkg}: {result.hint}")
5671:     """
5672:     results = {}
5673:
5674:     # Known native dependencies
5675:     native_deps = {
5676:         "pyarrow": ["zstd"],
5677:         "polars": [],  # Statically linked
5678:         "blake3": [],  # Statically linked
5679:     }
5680:
5681:     for package in packages:
5682:         # Check package itself
5683:         results[package] = check_wheel_compatibility(package)
```

```
5684:
5685:         # Check system libraries
5686:         if package in native_deps:
5687:             for lib in native_deps[package]:
5688:                 lib_result = check_system_library(lib)
5689:                 if not lib_result.available:
5690:                     results[f"{package}:{lib}"] = lib_result
5691:
5692:     return results
5693:
5694:
5695: def print_native_report() -> None:
5696:     """
5697:     Print a comprehensive native environment report.
5698:
5699:     This is useful for debugging environment issues.
5700:     """
5701:     print("=== Native Environment Report ===")
5702:     print(f"Platform: {platform.system()} {platform.release()}")
5703:     print(f"Architecture: {platform.machine()}")
5704:     print(f"Python: {sys.version}")
5705:     print()
5706:
5707:     print("CPU Features:")
5708:     cpu_result = check_cpu_features()
5709:     print(f"  {cpu_result.message}")
5710:     print()
5711:
5712:     print("FIPS Mode:")
5713:     print(f"  {'Enabled' if check_fips_mode() else 'Disabled'}")
5714:     print()
5715:
5716:     print("Critical Packages:")
5717:     packages = ["pyarrow", "polars", "blake3"]
5718:     results = verify_native_dependencies(packages)
5719:     for name, result in results.items():
5720:         status = "â\234\223" if result.available else "â\234\227"
5721:         print(f"  {status} {name}: {result.message}")
5722:         if result.hint:
5723:             print(f"     Hint: {result.hint}")
5724:     print()
5725:
5726:     print("System Libraries:")
5727:     for lib in ["zstd", "icu", "omp"]:
5728:         result = check_system_library(lib)
5729:         status = "â\234\223" if result.available else "â\234\227"
5730:         print(f"  {status} {lib}: {result.message}")
5731:         if result.hint:
5732:             print(f"     Hint: {result.hint}")
5733:
5734:
5735: if __name__ == "__main__":
5736:     print_native_report()
5737:
5738:
5739:
```

```
5740: ================================================================================
5741: FILE: src/farfan_pipeline/compat/safe_imports.py
5742: ================================================================================
5743:
5744: """
5745: Safe Import System - Deterministic, Auditable, Portable
5746:
5747: This module implements the core import safety layer for the SAAAAAA system.
5748: All imports in the codebase should use this pattern for optional dependencies,
5749: ensuring fail-fast behavior, clear error messages, and no graceful degradation.
5750:
5751: Design Principles:
5752: - No silent failures - imports either succeed completely or fail loudly
5753: - No graceful degradation - partial functionality is rejected
5754: - Deterministic behavior - same inputs always produce same outputs
5755: - Explicit error messages with installation hints
5756: - Support for alternative packages (e.g., tomllib vs tomli)
5757: """
5758:
5759: from __future__ import annotations
5760:
5761: import importlib
5762: import sys
5763: from typing import TYPE_CHECKING
5764:
5765: if TYPE_CHECKING:
5766:     import types
5767:
5768:
5769: class ImportErrorDetailed(ImportError):
5770:     """
5771:     Enhanced import error with context and actionable guidance.
5772:
5773:     This exception is raised when a required import fails and provides:
5774:     - The module that failed to import
5775:     - Installation instructions or hints
5776:     - Alternative packages if available
5777:     - Context about why the import is needed
5778:     """
5779:
5780:     def __init__(self, module_name: str, hint: str = "", install_cmd: str = "") -> None:
5781:         """
5782:         Initialize detailed import error.
5783:
5784:         Parameters
5785:         ----------
5786:         module_name : str
5787:             Name of the module that failed to import
5788:         hint : str, optional
5789:             Human-readable context about why this module is needed
5790:         install_cmd : str, optional
5791:             Installation command to resolve the missing dependency
5792:         """
5793:         parts = [f"Failed to import '{module_name}'"]
5794:
5795:         if hint:
```

```
            parts.append(f"Context: {hint}")

        if install_cmd:
            parts.append(f"Install with: {install_cmd}")

        message = ". ".join(parts)
        super().__init__(message)

        self.module_name = module_name
        self.hint = hint
        self.install_cmd = install_cmd


def try_import(
    modname: str,
    *,
    required: bool = False,
    hint: str = "",
    alt: str | None = None,
) -> types.ModuleType | None:
    """
    Attempt to import a module with explicit error handling and guidance.

    This function provides controlled import behavior with clear failure modes:
    - Required imports fail immediately with detailed errors
    - Optional imports log warnings and return None
    - Alternative packages are tried if primary fails
    - All failures include installation hints

    Parameters
    ----------
    modname : str
        The fully qualified module name to import (e.g., 'httpx', 'polars')
    required : bool, default=False
        If True, raises ImportErrorDetailed on failure
        If False, logs to stderr and returns None
    hint : str, default=""
        Human-readable guidance for resolving the import failure
        Should include installation command or extra flag
        Example: "Install extra 'http_signals' or set source=memory://"
    alt : str | None, default=None
        Alternative module to try if primary fails
        Example: 'tomli' as alternative to 'tomllib'

    Returns
    -------
    types.ModuleType | None
        The imported module if successful, None if optional and failed

    Raises
    ------
    ImportErrorDetailed
        When required=True and import fails (including alternatives)

    Examples
    --------
```

```
    >>> # Optional dependency with hint
    >>> httpx = try_import("httpx", required=False, hint="Install extra 'http_signals'")
    >>> if httpx is None:
    ...     # Use memory:// source instead
    ...     pass
    >>> # Required dependency
    >>> pyarrow = try_import("pyarrow", required=True, hint="Install core runtime")
    >>> # Version-specific with fallback
    >>> toml = try_import("tomllib", alt="tomli", required=True,
    ...                   hint="Python<3.11 needs 'tomli'")
    Notes
    -----
    - This function must NEVER silently substitute mock objects
    - Failure modes must be explicit and actionable
    - Import-time side effects in target modules are NOT controlled here
    - This is NOT for lazy loading - use separate lazy_import() for that
    """
    try:
        return importlib.import_module(modname)
    except Exception as primary_error:
        msg = f"[IMPORT] Failed '{modname}'"

        # Try alternative package if specified
        if alt:
            try:
                return importlib.import_module(alt)
            except Exception as alt_error:
                # Both primary and alternative failed
                combined_error = ImportErrorDetailed(
                    modname, hint=f"{hint}; alternative '{alt}' also failed"
                )
                combined_error.__cause__ = alt_error

                if required:
                    raise combined_error from primary_error
                else:
                    sys.stderr.write(f"{msg} (optional); alt also failed. {hint}\n")
                    return None

        # Required import failed - abort immediately
        if required:
            raise ImportErrorDetailed(modname, hint=hint) from primary_error

        # Optional dependency: log and defer failure to call site
        # This allows the module to load but fail when the feature is used
        sys.stderr.write(f"{msg} (optional). {hint}\n")
        return None


def check_import_available(modname: str) -> bool:
    """
    Check if a module can be imported without actually importing it.

```

```
    This is useful for feature flags and conditional code paths without
    triggering import-time side effects.

    Parameters
    ----------
    modname : str
        The fully qualified module name to check

    Returns
    -------
    bool
        True if module can be imported, False otherwise

    Examples
    --------
    >>> if check_import_available("polars"):
    ...     # Use polars backend
    ...     pass
    >>> else:
    ...     # Fall back to pandas
    ...     pass
    """
    try:
        spec = importlib.util.find_spec(modname)
        return spec is not None
    except (ImportError, ModuleNotFoundError, ValueError, AttributeError):
        return False


def get_import_version(modname: str) -> str | None:
    """
    Get the version of an installed module without importing it.

    This uses metadata inspection to avoid import-time side effects.

    Parameters
    ----------
    modname : str
        The module/package name

    Returns
    -------
    str | None
        Version string if available, None otherwise

    Examples
    --------
    >>> get_import_version("numpy")
    '1.26.4'
    """
    try:
        # Python 3.8+ has importlib.metadata
        from importlib.metadata import version  # type: ignore
        return version(modname)
    except Exception:
        return None
```

```
5964:
5965:
5966: # Cache for lazy-loaded modules to ensure deterministic re-import
5967: _lazy_cache: dict[str, types.ModuleType | None] = {}
5968:
5969:
5970: def lazy_import(modname: str, *, hint: str = "") -> types.ModuleType:
5971:     """
5972:     Lazy-load a module with memoization for deterministic behavior.
5973:
5974:     This is for import-time budget optimization on heavy modules.
5975:     Use this in functions that are called infrequently or in cold paths.
5976:
5977:     Parameters
5978:     ----------
5979:     modname : str
5980:         Module to lazy-load
5981:     hint : str, default=""
5982:         Installation hint if import fails
5983:
5984:     Returns
5985:     -------
5986:     types.ModuleType
5987:         The imported module (cached after first call)
5988:
5989:     Raises
5990:     ------
5991:     ImportErrorDetailed
5992:         If the module cannot be imported
5993:
5994:     Examples
5995:     --------
5996:     >>> def to_arrow(df):
5997:     ...     pa = lazy_import("pyarrow", hint="Install core runtime")
5998:     ...     return pa.table(df)
5999:
6000:     Notes
6001:     -----
6002:     - Memoization ensures the module is only loaded once
6003:     - Cache is module-global, not process-global
6004:     - This does NOT avoid import-time side effects, just defers them
6005:     """
6006:     if modname in _lazy_cache:
6007:         cached = _lazy_cache[modname]
6008:         if cached is None:
6009:             raise ImportErrorDetailed(f"[IMPORT] Module '{modname}' previously failed")
6010:         return cached
6011:
6012:     try:
6013:         mod = importlib.import_module(modname)
6014:         _lazy_cache[modname] = mod
6015:         return mod
6016:     except Exception as e:
6017:         _lazy_cache[modname] = None
6018:         msg = f"[IMPORT] Failed lazy import of '{modname}'"
6019:         if hint:
```

```
            msg += f". {hint}"
        raise ImportErrorDetailed(msg) from e


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

"""
Concurrency module for deterministic parallel execution.

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

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

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