Leveraging Large Language Models for Unstructured Claims Data Analysis

Appendix C. Technical Architecture and Methodology Document

LLM-Based Claims Analysis System

Project: Leveraging LLMs for Unstructured Claims Data Analysis
Version: 3.0
Date: December 16, 2025
Status: Implementation Documentation
Scope: Architecture of implemented prototype
Target Audience: Actuaries, developers, researchers, and system implementers

Authors

Richard Tran, MD, MSCS (MDSight, LLC) - Technical Architecture and Implementation
Robert Lieberthal, PhD (Lieberthal & Associates, LLC) - Project Leadership
Jawand Singh (William & Mary University) – User Interface
Vietbao Phan, PharmD (Thomas Jefferson University)
Elizabeth Sottung, PharmD (Thomas Jefferson University)

Organizations

MDSight, LLC (Technical Development)
Lieberthal & Associates, LLC (Project Management)
Thomas Jefferson University
William & Mary University

About the Casualty Actuarial Society (CAS)

For over 100 years, the Casualty Actuarial Society (CAS) has been the trusted global authority advancing the practice of property and casualty (P&C) actuarial science, with nearly 12,000 members worldwide who apply their expertise to help people, businesses, and communities unlock opportunities and thrive in a rapidly changing world. The CAS delivers top-tier credentialing, cutting-edge research, and dynamic learning and professional development, grounded in real-world application and powered by a vibrant global member community. The CAS equips actuaries to advance their careers and deliver innovative, trusted solutions to complex and emerging P&C challenges. Learn more at casact.org.

Contents

Executive Summary

This document describes the technical architecture and methodology of the LLM-based claims analysis system. The system consists of four Python scripts that process unstructured claims text to extract actuarial variables using Large Language Model APIs. The architecture emphasizes modularity, configurability, and extensibility to support both research validation and practical deployment.

The implemented system features a provider abstraction layer supporting multiple LLM backends, comprehensive infrastructure components for configuration management and logging, and a two-stage analysis pipeline that processes documents individually before aggregating results. The codebase follows clean architecture principles with clear separation of concerns between data processing, LLM interaction, and results generation.

PART I: CONCEPTUAL ARCHITECTURE

1. Overview

1.1. Document Purpose, Scope, and Audience

This Technical Architecture and Methodology Document (TAMD) specifies an LLM-based claims analysis system that extracts actuarial variables from unstructured claims data. The document serves as both a conceptual guide explaining architectural rationale and a practical implementation reference for system components, algorithms, and deployment.

The system addresses a fundamental actuarial challenge: most claims information exists in unstructured formats (medical records, adjuster notes, phone transcripts, settlement documents) that resist systematic analysis. While actuaries excel at analyzing structured numerical data, valuable predictive information in narrative text remains inaccessible or requires costly manual extraction. This prototype demonstrates how Large Language Models (LLM) can systematically convert unstructured claims text into standardized categorical variables for reserving, ratemaking, and claims management.

Target Audiences

- **Actuaries**: Conceptual explanations of two-stage processing, compound scoring, and variable extraction
- **Software Architects**: Processing pipeline specifications, algorithm implementations, data structure designs
- **Developers**: Infrastructure components, script implementations, configuration details
- **Regulatory Reviewers**: Conflict resolution protocols, provenance tracking, validation frameworks

The scope reflects the Casualty Actuarial Society research project "Leveraging LLMs in Unstructured Claims Data." The system consists of four Python scripts orchestrating a pipeline from synthetic data generation through final variable extraction, focusing on medical records as the primary use case. While targeting workers' compensation and property-casualty claims, the modular architecture supports extension to other insurance lines and document types.

1.2. System Overview

The Problem

The insurance industry generates enormous volumes of textual data throughout the claims lifecycle, yet this information remains largely inaccessible to actuarial analysis. Medical providers document clinical assessments in narrative notes. Claims adjusters record investigative findings. Phone conversations capture injury descriptions. Legal settlements document resolution terms. Each contains signals valuable for predicting costs and stratifying reserves, but traditional actuarial methods cannot systematically access this information at scale.

Current practice relies on manual review by specialists who assign categorical variables based on professional judgment. While producing high-quality assessments for individual claims, this approach suffers three critical limitations: (1) inconsistency across reviewers, (2) resource constraints preventing comprehensive coverage, and (3) incompleteness where many documents receive no review. The result is systematic underutilization of available information and missed opportunities for enhanced actuarial precision.

The Solution

This system demonstrates how LLMs bridge the gap between unstructured claims text and structured actuarial variables. LLMs possess natural language understanding capabilities enabling them to read narrative documents, interpret specialized terminology, and extract categorical assessments. When properly architected with validation and conflict resolution mechanisms, LLMs perform large-scale document analysis with consistency and completeness that manual review cannot achieve.

The solution processes claims through a four-script modular pipeline. Scripts 1-2 generate realistic synthetic data for testing, converting FHIR medical records to JSON and augmenting with diverse document types. Script 3 preprocesses real-world text files into structured JSON. Script 4 performs core analysis: extracting document-specific features in Stage 1, then aggregating across documents to produce final claim-level actuarial variables in Stage 2.

This two-stage architecture reflects real-world claims characteristics. Documents arrive at different times, carry different authority levels, and may contain conflicts. The system processes each document independently in Stage 1 using type-specific extraction logic, then systematically resolves conflicts in Stage 2 through compound scoring that weights sources by document authority, extraction confidence, temporal recency, and variable-specific importance.

Output Structure

Structured JSON containing core actuarial variables organized into three categories: (1) reserving variables (injury severity, medical complexity, treatment type, development pattern, ultimate cost), (2) ratemaking variables (risk level, causation type, industry risk, safety compliance, experience modifier impact), and (3) claims management variables (litigation risk, settlement likelihood, management complexity, fraud risk, claimant cooperation). Each variable includes confidence scores, rationale for the assigned value, and provenance metadata tracing to source documents.

Key Capabilities

- **Document Specialization:** Type-specific extraction logic ensures medical records, legal documents, and settlement agreements are processed with appropriate terminology and context.
- **Comprehensive Document Analysis**: The system processes each document independently to extract features without bias, then synthesizes information across all documents to detect inconsistencies, track injury progression, and resolve conflicts through compound scoring that prioritizes authoritative, recent, and reliable sources.
- **Temporal Awareness:** Recency weighting ensures recent information receives appropriate priority while preserving historical perspective.
- **Transparency and Auditability:** Every extracted variable links to source documents with supporting text. Compound score breakdowns show conflict prioritization. Confidence scores indicate assessment certainty.
- **Extensibility:** New document types, actuarial variables, and LLM providers can be added through configuration without architectural redesign. Organizations customize document weights, temporal decay, and confidence thresholds through YAML files.

1.3. Design Principles

Three foundational principles guide all architectural decisions and ensure the solution addresses real-world actuarial requirements.

- **Separation of Concerns:** The architecture maintains clear boundaries between functional responsibilities. Data generation (Scripts 1-2) operates independently from analysis (Scripts 3-4). LLM provider abstraction decouples business logic from API implementations. Configuration

through YAML files separates parameters from code. Stage boundaries separate document-level extraction from claim-level aggregation. This enables parallel development, provider switching without code changes, configuration updates without releases, and independent testing of pipeline stages.

- **Extensibility:** The system anticipates organizational diversity in document types, variable definitions, and requirements. The LLM provider abstraction enables adding new LLM backends through a single interface. Document-type-specific processing allows adding new formats by defining extraction schemas. Variable definitions in YAML enable custom variables and feature importance mappings through configuration. Template-based prompts support organizational terminology and classification schemes. Configurable scoring components allow adjusting document weights and temporal decay to match claim characteristics.
- **Research Focus:** As a research prototype, the system prioritizes transparency, measurability, and reproducibility over production optimization. Comprehensive logging captures API interactions, prompts, responses, and decisions for detailed behavior analysis. Cost and performance tracking provides quantitative data for feasibility assessment. Synthetic data generation creates reproducible test cases with known ground truth. Function calling ensures schema compliance. Explicit compound scores enable validation of prioritization logic. Confidence scores provide quantitative reliability indicators. This research-first approach produces a system actuaries can evaluate, validate, and understand before production deployment.

1.4. Document Organization

This document is organized into two parts with different formats and purposes, supplemented by comprehensive appendices.

- **Part I: Conceptual Architecture** (Sections 1-5) explains design philosophy, algorithmic approaches, and data structures in prose format. Addresses: Why this architecture? How does compound scoring work? What drives variable taxonomy? Targets architects, actuarial reviewers, and technical leaders evaluating the system's foundation or actuarial validity.
- **Part II: Implementation Reference** (Sections 6-10) provides component specifications, configurations, and deployment details in hybrid format (prose intros + bullets/tables). Answers: How is configuration structured? What classes implement functionality? How are scripts deployed? What are extension points? Targets developers, operations teams, and technical staff implementing or troubleshooting the system.
- **Appendices** (A-D) contain complete reference material: full configuration examples with documentation, complete JSON schemas, algorithm pseudocode, and code repository guides. Serve as quick reference for specific technical details.

Navigation Guide:

- **For conceptual understanding:** Continue to Section 2 (Architecture Design Philosophy)
- **For processing methodology:** Go to Section 3 (Core Processing Methodology)
- **For implementation details:** Skip to Part II, Section 6 (Core Infrastructure Components)
- **For variable specifications:** See Section 5.2 (Actuarial Variable Taxonomy)
- **For quick reference:** Consult Appendices A-D

This introduction establishes the foundation for understanding the LLM-based claims analysis system. The following sections provide progressively more detailed specifications of the architecture, algorithms, and implementation approaches that enable systematic extraction of actuarial variables from unstructured claims data.

2. Architecture Design Philosophy

2.1. Design Principles

The system architecture balances competing requirements through three core principles that address both actuarial needs and practical implementation constraints.

- **Separation of Concerns** maintains clear functional boundaries enabling independent development and deployment. Data generation (Scripts 1-2) operates independently from analysis (Scripts 3-4). The LLM provider abstraction decouples business logic from APIs, changing providers requires implementing a single interface rather than modifying extraction algorithms. YAML configuration separates tunable parameters from code, enabling actuaries to adjust document weights and thresholds without software releases. Stage boundaries between document and claim processing make each phase independently testable.

  This delivers tangible benefits: teams work on scripts simultaneously without coordination, testing targets specific stages without full pipeline execution, and organizations deploy components progressively rather than requiring complete system implementation.

- **Extensibility** anticipates organizational diversity in documents, variables, and requirements. The provider factory pattern enables adding LLM backends by implementing the LLMProvider interface. Document-type-specific processing allows adding new formats through YAML schema definitions. Variable definitions specify custom actuarial variables and feature importance mappings through configuration files editable by actuaries, not developers.

  Template-based prompts support organizational terminology. The compound scoring formula exposes configurable components for organization-specific weighting. This ensures the system adapts to organizational context while preserving architectural consistency.

- **Research Focus** prioritizes transparency, measurability, and reproducibility. Comprehensive logging captures API interactions, prompts, responses, and decisions for behavior analysis. Cost and performance tracking provides quantitative data for feasibility assessment. Synthetic data generation creates reproducible test cases with known ground truth.

  Design choices favor explainability: function calling enforces schema compliance, compound scores are explicitly calculated and logged, and confidence scores accompany all variables. This produces a system actuaries can evaluate and understand before production deployment.

2.2. Two-Stage Processing Rationale

The two-stage architecture (document-level extraction, then cross-document aggregation) addresses seven fundamental challenges that single-pass architectures cannot resolve.

- **Document Specialization:** Different document types require fundamentally different extraction approaches. Medical records use clinical terminology and structured formats. Adjuster notes contain investigative findings and settlement evaluations. Phone transcripts capture conversational language with limited structure. A generic extraction prompt would either miss type-specific information or become unmaintainable.

  The system employs a two-tier template architecture: category templates provide broad extraction frameworks for document families (medical, legal, investigation, workplace, financial, external), while specialized templates refine extraction for specific document types. This enables the system to handle diverse document types, even those without specialized templates, by falling back to appropriate category-level extraction logic. For example, while the system has specialized templates for the six core document types (phone transcripts, medical provider letters, clinical notes, adjuster notes, settlement notes, claimant statements), the category system can process dozens of additional document types like IME reports, surveillance reports, court filings, and OSHA documents using category-appropriate extraction schemas.

- **Temporal Evolution Tracking:** Claims evolve substantially over their lifecycle. Single-pass processing must choose one assessment, discarding temporal information. Stage 1 preserves each document's perspective with explicit timestamps. Stage 2 synthesizes this temporal sequence, identifying when severity changes (progression or improvement), detecting unexpected developments, and documenting complete evolution. This temporal awareness is valuable for reserving—knowing whether claims deteriorate or improve affects reserve adequacy.
- **Source Reliability Weighting:** Not all documents carry equal authority. Medical provider letters contain professional clinical assessments. Phone transcripts capture preliminary reports. Claimant statements provide subjective perspectives. Actuarial analysis requires prioritizing authoritative sources while retaining lower-confidence information.

  Two-stage architecture enables explicit prioritization through document weight assignments (0.6-1.0 range). Medical provider letters: 1.0 (highest professional clinical assessment). Settlement adjuster notes: 0.9 (authoritative evaluation). Clinical notes: 0.85 (professional documentation). Adjuster notes initial: 0.7 (preliminary investigation). Phone transcripts and claimant statements: 0.6-0.7 (initial/subjective reports).
- **Auditability and Transparency:** Actuarial analysis demands traceability. When the system determines injury severity as "Major," actuaries need documentation: which documents supported this, what evidence was cited, how conflicts were resolved, and what confidence level applies. Regulatory compliance requires comprehensive audit trails from source text to final variables.

  Stage 1 creates per-document extractions as audit records. Each variable extraction includes source document ID, confidence score, supporting text snippet, and extraction rationale. Stage 2 aggregations reference Stage 1 sources, creating complete provenance chains. An actuary reviewing "Major" severity traces back through settlement documentation, medical records detailing treatment, and initial assessments showing progression.
- **Flexibility and Extensibility:** Insurance organizations have diverse document types, variable definitions, and analytical requirements. Monolithic architectures resist customization. Two-stage design provides multiple extension points without system-wide modifications.

  New document types integrate by creating Stage 1 schemas and prompts without modifying Stage 2 logic. New actuarial variables integrate by updating Stage 2 schemas without changing Stage 1 extractions. Organizations customize document weights, confidence thresholds, and conflict resolution through configuration files without code changes.
- **Scalability and Performance:** Stage 1 processes documents sequentially but independently. Each document is analyzed in isolation without awareness of other documents in the claim. This ensures that each document's features are extracted based solely on its own content, preventing information bleeding between documents and maintaining clean document-level extractions.

  Stage 2 aggregation adds minimal overhead as lightweight rule-based logic. Organizations can scale Stage 1 processing (expensive LLM operations) independently from Stage 2 synthesis (fast deterministic logic), deploying multiple Stage 1 workers feeding a single Stage 2 aggregator.
- **Balance of Competing Objectives:** Claims analysis requires balancing contradictory requirements: preserving document-specific nuances while producing unified assessments, respecting source authority while remaining sensitive to new evidence, maintaining extraction independence while leveraging cross-document context.

  Two-stage architecture explicitly separates these concerns. Stage 1 focuses on faithful extraction from individual sources, maximizing document-specific accuracy. Stage 2 focuses on intelligent

synthesis across sources, maximizing aggregate reliability. This separation allows optimizing each stage for its specific objectives without compromise.

2.3. System Data Flow

The system processes claims through a clear pipeline with distinct stages, each performing specific transformations.

- **Input Layer** accepts two formats. For synthetic data testing, FHIR bundles provide realistic medical records with standardized structure. For real-world analysis, plain text files contain actual claims documents (medical records, adjuster notes, transcripts, settlements). Both formats enter through standardized preprocessing: normalizing encoding, extracting metadata, and structuring content.
- **Stage 1: Document Processing** analyzes each document independently to extract document-specific features. The system identifies document type through filename conventions or content classification. Document-type-specific extraction loads appropriate schemas and prompts. The LLM extracts relevant features with confidence scores. Temporal metadata records document dates, processing timestamps, and timeline positions. Each document produces Stage 1 JSON containing features, confidence scores, text snippets, and metadata.
- **Stage 2: Claim Synthesis** aggregates all Stage 1 outputs for a claim to produce final actuarial variables. The system loads all Stage 1 JSON files (typically 5-10 documents). For each actuarial variable, it collects all related document-level extractions. Compound scores are calculated combining document weight, extraction confidence, temporal recency, and feature importance. Sources are ranked by compound score. The Stage 2 LLM receives ranked sources with score explanations and synthesizes final values with rationale documentation.

  Conflict resolution occurs systematically through compound score prioritization. When documents provide different severity values (mild vs. moderate), the system calculates which source is most reliable based on document authority, extraction confidence, recency, and variable-specific appropriateness. Resolution logic is explicit, logged, and auditable.
- **Output Layer** produces structured JSON containing claim-level actuarial variables by functional category. **Reserving** variables (injury severity, medical complexity, treatment type, development pattern, ultimate cost) support reserve estimation. **Ratemaking** variables (risk level, causation type, industry risk, safety compliance, experience modifier impact) inform pricing. **Claims management** variables (litigation risk, settlement likelihood, management complexity, fraud risk, claimant cooperation) guide handling strategy. Each variable includes assessed value, aggregate confidence score, source document list, and synthesis rationale.

3. Core Processing Methodology

This section specifies the processing algorithms, scoring systems, and conflict resolution mechanisms that enable systematic extraction and aggregation of actuarial variables from unstructured claims data.

3.1. Stage 1: Document-Level Extraction

Stage 1 processes each document independently to extract document-specific features with confidence scores and supporting evidence.

Purpose and Goals: Stage 1 achieves faithful extraction from individual sources, maximizing document-specific accuracy without concern for cross-document conflicts. Each document receives specialized processing appropriate to its type and terminology.

Processing Steps:

1. **Document Type Recognition:** The system identifies the type of document through the native "document_type" field within the FHIR Json bundle or via filename conventions (<initials>_<claim_id>_<document_type>.txt). Pre-defined mappings and intelligent keyword-based matching map these document types to a three-tier template hierarchy.
    - **Tier 1 - Specialized Templates**: clinical_note, medical_provider_letter, phone_transcript, adjuster_notes_initial, settlement_adjuster_notes, claimant_statement
    - **Tier 2 - Category Templates**: medical, legal, investigation, workplace, financial, external
    - **Tier 3 - Universal Fallback Template**: Handles any document type not covered by specialized or category templates
2. **Schema Loading:** Each document type uses two specialized components that implement the document-specific processing described in section 2.2. (1) Specialized extraction templates provide document-type-specific instructions to guide LLM extraction. (2) Function Call JSON Schemas further define structured output format for validation, specifying field names, data types, and required elements for each document type.
3. **Prompt Construction:** Document-type-specific prompts emphasize relevant information based on the selected template. Each prompt embeds variable definitions providing the LLM with extraction criteria specific to that document type.
4. **LLM Extraction:** Function calling with specialized schemas ensures structured output conforming to predefined JSON schemas. The LLM extracts all relevant variables for the document type, focusing on information that document type typically contains. Not all documents contribute to all variables - extraction is selective based on document type appropriateness.
5. **Confidence Scoring:** Each extracted variable receives a confidence score (0.0-1.0) reflecting extraction certainty based on text clarity and evidence presence. Explicit statements receive high confidence (0.9-1.0). Strong inferences score 0.7-0.89. Moderate inferences score 0.5-0.69. Weak or speculative extractions score below 0.5.
6. **Fallback Hierarchy:** When confidence scores are missing, the system applies granular defaults: variable-specific default from definitions, document-type default from configuration, or program-level default (0.5, configurable via YAML).
7. **Temporal Metadata Capture:** System records document date (extracted from text using regex patterns), processing timestamp, and document age in days. This metadata enables temporal weighting in Stage 2.
8. **Source Text Extraction:** Supporting rationales are captured for each variable, which may include text references, paraphrased evidence, or extraction reasoning, creating audit trails from final values back to source documents.

Output: Each claim produces Stage 1 JSON containing extracted features, confidence scores, extraction rationale, source documents and their metadata (date, age, weight, recency factor).

3.2. Stage 2: Cross-Document Aggregation

Stage 2 synthesizes all Stage 1 outputs for a claim into final actuarial variables, resolving conflicts through systematic prioritization.

Purpose and Goals: Stage 2 achieves intelligent synthesis across sources, maximizing aggregate reliability by systematically evaluating conflicting information and producing unified assessments with comprehensive provenance.

Processing Steps:

1. **Input Assembly:** System loads all Stage 1 JSON files for the claim (typically 5-10 documents spanning the claim lifecycle) and validates structural completeness.
2. **Feature Grouping:** Stage 1 features are grouped by Stage 2 variable name. Multiple Stage 1 features may map to one Stage 2 variable—settlement_amount and total_medical_cost from different document types both contribute to ultimate_cost_prediction.
3. **Compound Score Calculation:** For each feature within a variable group, calculate compound score (detailed in Section 3.3). This produces ranked source lists for each final variable.
4. **Source Ranking:** Within each variable group, sources are ranked highest to lowest by compound score. The ranking explicitly shows document authority, extraction confidence, temporal recency, and variable-specific importance.
5. **Prompt Construction:** Stage 2 prompt displays ranked sources with complete score breakdowns. The LLM receives guidance to prefer higher-scored sources but may override with documented rationale if business logic or domain knowledge suggests otherwise.
6. **LLM Aggregation:** Using Stage 2 schema (claim-level variables), the LLM synthesizes final values. When sources agree, aggregation is straightforward. When sources conflict, the LLM considers compound scores, temporal evolution, and actuarial logic to determine final values.
7. **Rationale Documentation:** For each variable, the LLM documents which sources contributed, how conflicts were resolved, and reasoning for the final assessment. This creates transparent audit trails.
8. **Provenance Generation:** Output includes complete source lists linking each final variable to contributing Stage 1 documents, enabling traceability from final assessments back to original text.

Output: Claim-level JSON containing core actuarial variables organized by category (reserving, ratemaking, claims management), each with value, confidence, source list, and rationale. The output also includes claim-level quality flags for manual review indicators.

3.3. Compound Scoring System

The compound scoring system provides systematic prioritization of conflicting information through a four-component weighted formula addressing the fundamental challenge: when different sources provide conflicting values, how do we decide which to trust?

Mathematical Formulation:

- **compound_score** = **document_weight** × **confidence** × **recency** × **feature_importance**

Where each component is normalized to [0.0, 1.0], ensuring compound scores fall within [0.0, 1.0]. Higher scores indicate more reliable sources for the specific variable.

3.3.1. Component 1: Document Weight (0.6-1.0)

Document weight reflects inherent authority and reliability of each document type. These weights are configured in YAML and can be customized organizationally.

Document Type	Weight	Rationale
Medical Provider Letters	0.95-1.0	Professional medical assessment, highest clinical authority
Settlement Adjuster Notes	0.85-0.95	Final authoritative source at claim resolution, experienced professional assessment
Clinical Notes	0.85-0.9	Professional clinical observations and diagnosis
Adjuster Notes (Initial)	0.75-0.85	Preliminary professional assessment and investigation
Phone Transcripts	0.6-0.7	Initial report, limited verification
Claimant Statements	0.6-0.65	Subjective self-report

Implementation: Weights are read from config.yaml under document_types section. Missing weights default to 0.7 (moderate reliability).

3.3.2. Component 2: Confidence Score (0.0-1.0)

The confidence scoring system is highly configurable and represents the LLM's certainty in Stage 1 extraction, based on text clarity, supporting evidence, internal consistency, and document quality.

Interpretation Scale:

- **0.9-1.0:** Explicit statement - value clearly stated with minimal interpretation
- **0.7-0.89:** Strong inference - value implied with high certainty
- **0.5-0.69:** Moderate inference - value derived from context with reasonable certainty
- **0.3-0.49:** Weak inference - value suggested but not clearly supported
- **0.0-0.29:** Speculative - value highly uncertain or guessed

Generation: During Stage 1, the LLM is explicitly prompted to provide confidence scores alongside extracted values, considering whether the value is explicit or inferred, quality of supporting text, ambiguity level, and information specificity.

Fallback Handling: When confidence scores are missing: (1) feature-specific default if defined in variable definitions, (2) document-type default if defined in configuration, (3) program-level default of 0.5.

3.3.3. Component 3: Recency Factor (0.6-1.0)

Recency factor applies temporal decay to document age, reflecting that more recent information is generally more reliable. Claims evolve - medical conditions change, cost estimates become outdated, assessments are refined.

Stepped Decay Function

Document Age	Recency Factor	Rationale
≤ 30 days	1.0	Recent, fully reliable
≤ 90 days	0.9	Medium-term, highly reliable
≤ 180 days	0.8	Aging, still reliable
≤ 365 days	0.7	Old, moderately reliable
> 365 days	0.6	Very old, less reliable

Document-Specific Overrides: Settlement adjuster notes have a fixed recency factor regardless of age. Settlement amounts and legal determinations remain final and don't decay. Phone transcripts use accelerated decay as initial reports rapidly lose relevance.

Edge Case Handling:

- **Missing dates:** Default to factor 0.7 (conservative)
- **Future dates:** Treated as current (factor 1.0) and flagged for review
- **Invalid formats:** Log warning and apply missing date default

3.3.4. Component 4: Feature Importance (0.0-1.0)

Feature importance captures variable-specific source preferences. For some actuarial variables, certain document types are inherently more authoritative independent of document age or general reliability.

Importance Guidelines by Value Type:

- **Actual/Final values (0.95-1.0):** Confirmed amounts from settlement or final medical bills
- **Professional assessments (0.85-0.95):** Expert opinions from medical providers or experienced adjusters
- **Calculated estimates (0.7-0.85):** Derived projections based on available data
- **Preliminary estimates (0.5-0.7):** Early projections subject to revision
- **Initial reports (0.3-0.5):** First mentions before full investigation

Definition and Configuration: Feature importance values are defined in the actuarial variable definitions YAML file. Each Stage 2 variable has a feature_importance section mapping document types and their Stage 1 features to importance scores.

Example: Ultimate Cost Prediction

```
yaml
ultimate_cost_prediction:
  feature_importance:
    settlement_adjuster_notes.settlement_amount: 1.0
    medical_provider_letter.total_medical_cost: 0.90
    adjuster_notes_initial.initial_reserve_amount: 0.55
```

The example shows a simplified subset of mappings for ultimate cost prediction, which indicates that settlement amounts are most important (1.0), while cost mentions in initial adjuster note is less important (0.55).

3.3.5. Complete Calculation Example (Hypothetical Scenario)

Scenario: Ultimate cost prediction with three conflicting sources.

Source 1: Settlement Adjuster Notes

- Document weight: 0.7 (settlement_adjuster_notes)
- Confidence: 0.95 (explicitly stated)
- Recency: 1.0 (15 days old)
- Feature importance: 1.0 (settlement_amount)
- **Compound score**: 0.7 × 0.95 × 1.0 × 1.0 = 0.6650

Source 2: Medical Provider Letter

- Document weight: 1.0 (medical_provider_letter)
- Confidence: 0.85 (professional assessment)
- Recency: 0.9 (60 days old)
- Feature importance: 0.90 (total_medical_cost)
- **Compound score**: 1.0 × 0.85 × 0.9 × 0.90 = 0.6885

Source 3: Phone Transcript

- Document weight: 0.9 (phone_transcript)
- Confidence: 0.60 (vague mention)
- Recency: 0.8 (150 days old)
- Feature importance: 0.40 (hospital_er_treatment)
- **Compound score**: 0.9 × 0.60 × 0.8 × 0.40 = 0.1728

Source Ranking:

- **Medical Provider Letter**: 0.6885 (highest priority)
- **Settlement Adjuster Notes**: 0.6650 (secondary source)
- **Phone Transcript**: 0.1728 (low priority)

Despite settlement notes being the final authoritative source, the Medical Provider Letter receives a slightly higher compound score (0.6885 vs 0.6650) due to its higher document weight (1.0 vs 0.7).

In Stage 2 aggregation, the LLM receives these sources ranked by compound score with complete breakdowns. The prompt instructs preferring higher-scored sources but allows override with documented rationale if conflicting information reveals errors.

Benefits:

- **Improved Conflict Resolution:** Systematic prioritization reduces arbitrary decisions
- **Increased Transparency:** Full score breakdown visible in logs and prompts
- **Better Accuracy:** Recent information and authoritative sources preferred
- **Maintained Flexibility:** LLM can override with explanation
- **Minimal Performance Impact:** Adds minimal processing overhead with simple arithmetic

3.4. Temporal Weighting Algorithm

Temporal weighting systematically accounts for document age in compound scoring through the recency factor calculation.

- **Stepped Decay Implementation:** The system uses stepped decay rather than continuous exponential decay for interpretability, configurability (organizations define custom thresholds), and stability (small date differences don't cause score fluctuations).

Configuration Parameters:

```
yaml
temporal_weighting:
  enabled: true
  decay_function: "stepped"   # or "exponential"
  stepped_decay:
    thresholds:
      - days: 30, factor: 1.0
      - days: 90, factor: 0.9
      - days: 180, factor: 0.8
      - days: 365, factor: 0.7
      - days: 999999, factor: 0.6
  missing_date_factor: 0.7
  document_overrides:
    settlement_adjuster_notes:
      decay_function: "none"
      fixed_factor: 1.0
```

Date Extraction: The DocumentDateExtractor class handles date extraction using regex patterns (ISO, US, written formats), metadata parsing, and filename extraction. When multiple dates are found, it selects the latest date. Missing dates default to factor 0.7.

3.5. Conflict Resolution Protocol

When different documents provide conflicting values for the same variable, the system applies systematic conflict resolution guided by compound scores.

Protocol Steps:

1. **Conflict Detection:** Group extractions by variable name and compare values. Flag differences as conflicts.
2. **Priority-Based Resolution:** Rank sources by compound score. Present to Stage 2 LLM with explicit ranking and score breakdowns.
3. **Rationale Documentation:** LLM documents selected value and resolution reasoning. If overriding top-ranked source, LLM must explain why.
4. **Provenance Recording:** Final output includes all sources, their scores, and resolution explanation.

Resolution Rules (implicit in scoring):

- Recent information > Older information (temporal evolution)
- Actual costs > Projected costs (realized vs. estimated)
- Settlement documents > Initial assessments (final authoritative)
- Objective measurements > Subjective assessments (tests vs. patient reports)
- High confidence > Low confidence (clear evidence vs. ambiguous)

Override Handling: The Stage 2 LLM may override compound score rankings when business logic requires (actual value supersedes estimate), clear errors detected (obvious contradiction), multiple lower sources agree against highest, or domain knowledge suggests otherwise. Overrides require rationale documentation.

4. Document Processing Pipeline

Document processing prepares unstructured claims text for LLM analysis. The current implementation uses minimal text normalization focused on basic validation rather than sophisticated transformation.

4.1. Text Normalization

Text normalization in the current pipeline is intentionally minimal to preserve original document content for LLM analysi.

- **UTF-8 Encoding Handling:** All text files are read with UTF-8 encoding specified in the configuration file with configuration parameter, text_file_processing.encoding (default: utf-8). Files that cannot be decoded with the specified encoding will raise standard Python UnicodeDecodeError exceptions
  - **Current Limitation:** No encoding detection or fallback mechanism exists. Files with non-UTF-8 encoding must be converted externally before processing.
- **Whitespace Handling:** Whitespace is normalized while preserving clinically significant formatting. Leading and trailing whitespaces are removed. Internal whitespaces (spaces, tabs, line breaks) are preserved.
  - **Rationale**: LLMs can handle variable whitespace, and preserving original formatting may contain semantic information (e.g., indentation in medical notes, table structures).
- **Content Preservation:** All characters within the document content are preserved without modification. Medical symbols (°, ±, %, $), special characters, and punctuation are maintained in their original form.
  - **Rationale**: Modern LLMs are robust to varied character representations and can interpret symbols contextually.
- **Minimum Content Validation:** Files must meet a configurable character count threshold (default: 50 characters) before processing. Files below this threshold are skipped and logged as warnings. Character count validation occurs after .strip() removes leading and trailing whitespace, excluding whitespace-only files.

4.2. Metadata Extraction

Metadata extraction pulls structured information from filenames and document content to enable proper document identification, classification, and temporal analysis.

- **Filename Parsing Logic:** The system expects filenames following standardized convention: <patient_initials>_<claim_id>_<document_type>.txt

  Examples: JD_CLM12345_medical_provider_letter.txt

  Parsing extracts: patient initials (first segment), claim ID (second segment), document type (remaining segments joined with underscores). Document type is mapped through configuration (e.g., "medical" → "medical_provider_letter"). When parsing fails (fewer than 3 segments), the file is logged in processing report as 'Invalid filename format' and skipped entirely. it does not proceed to Stage 1 or fallback classification.
- **Document Type Classification:** When document type cannot be determined from filename, the system employs intelligent content-based classification using the three-tier template hierarchy (specialized, category, and universal fallback templates) described in Section 3.1.
- **Intelligent Keyword Matching:** Priority-based matching resolves ambiguous types. Settlement references with "adjuster" match settlement_adjuster_notes (priority 1). Medical/provider/doctor/physician with letter/report/note/summary match medical_provider_letter (priority 2). Adjuster with notes/investigation matches

adjuster_notes_initial (priority 3). System iterates through rules in priority order, selecting first match.

- **Date Detection and Parsing:** Document dates are extracted exclusively from document content (not filenames) using regex pattern matching. The ClassExtractDate class scans full document text for date patterns.

  Supported Formats:
  - ISO format: 2024-10-15, 2024/10/15
  - US format: 10/15/2024, 10-15-2024
  - Written format: October 15, 2024; Oct 15, 2024
  - Labeled dates: Date: 10/15/2024, Document Date: October 15, 2024

  Date Selection Logic: When multiple dates are found, the system selects the **most recent** date, assumming it represents the document's authoritative date. Documents without parseable dates receive the default temporal factor of 0.7 in Stage 2 compound scoring.

Implementation Limitations
Current implementation does **not** support: date extraction from filenames, relative date expressions ("3 months ago", "last week"), multiple date conflict detection or plausibility validation, header-specific metadata extraction (patient IDs, provider names), or probabilistic classification confidence scoring. The simplified approach prioritizes reliability (pattern-based extraction is predictable and reproducible), transparency (explicit matching rules and date selection logic), and efficiency (single-pass regex scanning). Future enhancements could add sophisticated date parsing, conflict detection, and header extraction without architectural changes.

4.3. Quality Assessment

Quality assessment filters unusable documents before LLM processing through gatekeeping checks, balancing resource efficiency with permissive inclusion of varied content.

Assessment Philosophy
The system applies light-touch quality screening that excludes only obvious unusable content- empty files, corrupted text, or fragments too short for semantic analysis. Questionable documents are processed rather than rejected, leveraging LLM robustness to handle input variability. This approach shifts quality control from preventive (input filtering) to detective (output validation), recognizing that extraction quality is difficult to predict from input characteristics alone.

Quality Checks

- **Minimum Content Validation:** Files must contain at least 50 characters (configurable) after whitespace stripping. This threshold excludes empty or near-empty files while accepting terse content like brief adjuster notes. Files below threshold are skipped with warnings logged to processing reports.
- **Filename Validation:** Files must follow {patient_initials}_{claim_id}_{document_type}.txt convention for metadata extraction. Invalid formats are skipped since claim ID and document type cannot be determined.
- **Processing Report Generation**: Each run generates a JSON report tracking total files found, files processed, files skipped with reasons, and unified claim JSON files generated. This enables quality monitoring and threshold tuning based on operational experience.

Quality Strategy

- **Quality thresholds** are deliberately permissive. The 50-character minimum represents approximately one sentence. No readability metrics, sentence detection, or structure validation are applied since these would add complexity while providing limited value given LLM robustness

to input variability. LLMs successfully interpret documents with inconsistent formatting, abbreviations, mixed content types, and unconventional structure.

- **Failed extractions** due to poor input quality are not wasted. Stage 1 processing generates confidence scores indicating extraction uncertainty, enabling downstream filtering based on actual extraction quality rather than predicted input quality.

Relationship to Validation

Quality Assessment and Validation Framework (Section 5.5) serve complementary purposes:

- **Quality Assessment**: Input fitness evaluation before processing using simple heuristics (character counts, filename patterns)
- **Validation**: Output correctness verification after processing using sophisticated checks (schema conformance, business logic, actuarial consistency)

Quality assessment prevents waste by filtering unusable inputs. Validation ensures correctness by detecting extraction errors and logical inconsistencies. See Section 5.5 for comprehensive validation framework details.

Implementation Notes

- **Validation Performance**: All validation operations are lightweight and add minimal processing overhead. Document ingestion, schema validation, and structural checks add negligible overhead. Actuarial consistency checks complete efficiently due to dictionary lookups and targeted rule evaluation. Specific timing measurements should be conducted during production deployment to establish precise performance baselines.
- **Quality Flag Aggregation**: Quality flags accumulate throughout processing and are included in final output JSON under processing_metadata.quality_flags. This enables downstream systems to filter or flag claims based on validation concerns, implement custom thresholds for automatic acceptance, route flagged claims for manual review, and analyze patterns in validation issues to improve extraction prompts.
- **Future Enhancements**: The validation framework could be extended to support configurable validation severity levels (error, warning, info) with different handling strategies, statistical analysis of validation flag patterns to identify systematic issues, automated prompt tuning based on common validation failures, and integration with external validation services for domain-specific rule checking. However, these enhancements should only be implemented if empirical analysis shows they would improve operational efficiency or output quality.

5. Data Structures & Schema Design

Data structures and schemas formalize system inputs, outputs, and internal representations, enabling automated validation, consistent integration, and reliable data exchange between pipeline stages and external actuarial systems.

5.1. Schema Architecture Overview

The system employs a three-layer schema architecture corresponding to pipeline transformation stages. Each layer serves distinct validation and processing purposes while maintaining semantic consistency across transformations..

- **Layer 1: Input Validation Schemas** Input Validation Schemas verify raw input conformance before processing. Text input schemas validate filename conventions, encoding formats, and minimum content requirements. Document dates are extracted during Stage 1 processing via regex pattern matching (ClassExtractDate), not validated as input. Claim metadata schemas verify required identifiers (encounter_id, document_type, document_text) and organizational context. This layer rejects malformed inputs early, preventing downstream errors and resource waste.
- **Layer 2: Stage 1 Document-Level Schemas** define extraction outputs for each document type. Medical provider letter schemas specify clinical variable structures. Adjuster notes schemas define investigation variables. Clinical note schemas capture treatment patterns. Settlement adjuster note schemas formalize settlement evaluation. Phone transcript and claimant statement schemas document incident details and perspectives. Schema specialization by document type enables targeted extraction while maintaining consistent metadata structures across types.
- **Layer 3: Stage 2 Claim-Level Schema** defines final actuarial variable outputs aggregated from multiple documents. The schema organizes variables into reserving, ratemaking, and claims management categories (see Section 5.2). Each variable includes assessed value, confidence score, Stage 1 source references, and aggregation rationale. The schema provides rollup views for cost predictions and risk assessment, along with an overall claim narrative. Quality metrics and processing flags are maintained at the claim level. This schema serves as the integration contract with external actuarial systems.
- **Schema Organization Philosophy** Schemas enforce separation of concerns - input validation is independent of extraction logic, document-level extraction is independent of claim-level aggregation. This modularity enables schema evolution without breaking dependent components. Adding a new document type requires only Layer 2 schema addition and corresponding function schema configuration; existing schemas remain unchanged. Function schemas are loaded dynamically from JSON files via ClassLoadSchemas, with caching for performance. Consuming systems should validate against the appropriate layer schema to ensure compatibility.

5.2. Actuarial Variable Taxonomy

The system extracts 36 actuarial variables organized into three functional categories. Variables use controlled vocabularies with enumerated values ensuring consistency and enabling statistical analysis.

Variable Counts by Category

Schema enforcement prevents variable duplication across categories and ensures outputs conform to actuarial system integration requirements (see Section 5.3).

Category	Variable Count	Schema Constraints
Reserving	17	maxProperties: 17, additionalProperties: false
Ratemaking	8	maxProperties: 8, additionalProperties: false
Claims Management	11	maxProperties: 11, additionalProperties: false
Total	36	Enforced via JSON Schema validation

Variable Specification Format

Each variable in extraction prompts follows this JSON structure:

```
json
{
  "variable_name": "injury_severity",
  "category": "RESERVING",
  "definition": "Clinical severity of the injury based on medical assessment",
  "data_type": "string",
  "enum": ["minor", "moderate", "major", "catastrophic"],
  "extraction_guidance": "Assess based on documented functional impairment …. ",
  "feature_importance": {
    "medical_provider_letter.injury_severity_assessment": 0.95,
    "adjuster_notes_initial.injury_description": 0.80
  },
  "validation": {
    "min_confidence": 0.75,
    "rationale_required": true
  },
  "examples": {
    "minor": "Soft tissue strain, full recovery expected within weeks",
    "catastrophic": "Permanent total disability, ongoing care needs"
  }
}
```

This specification accompanies each prompt, ensuring LLMs extract variables consistently across documents. Feature importance mappings guide Stage 2 compound scoring (see Section 3.3).

5.2.1. Reserving Variables

Reserving variables inform loss reserve estimation and development pattern selection. These variables predict ultimate claim costs and settlement timing. Category contains 17 variables.

Variable	Definition	Data Type	Valid Values
Claim_severity	Overall claim severity assessment combining injury severity, treatment complexity, and financial impact	Categorical	minor, moderate, major, catastrophic
Injury_severity	Clinical severity of the injury based on medical assessment	Categorical	minor, moderate, major, catastrophic
Medical_complexity	Overall medical treatment complexity level Simple=straightforward care, Moderate=standard protocols, Complex=surgical/specialists, Highly_Complex=chronic/permanent disability care	Categorical	simple, moderate, complex, highly_complex
Treatment_type	Primary treatment approach for the injury	Categorical	conservative, surgical, ongoing_chronic, complex_rehab
Expected _development_pattern	Expected claim development timeline and pattern for actuarial modeling and loss reserving	Categorical	fast_resolution, average, slow_development, long_tail
Ultimate_cost_category	Categorical range of ultimate claim cost for loss development modeling. Must match ultimate_cost_prediction value via mathematical validation	Categorical	<25K, 25K-50K, 50K-100K, 100K-150K, >150K
Total_incurred_cost	Total costs paid and incurred to date	Numeric (USD)	≥0
Estimated_outstanding _cost	Estimated future costs to be incurred from current date until claim closure	Numeric (USD)	≥0
Ultimate_cost_prediction	Final predicted total claim cost including all medical, indemnity, and administrative expenses	Numeric (USD)	≥0
Permanent_disability _rating	Permanent partial or total disability rating if assessed	Numeric (%)	0-100
Recovery_timeline_days	Estimated or actual recovery period	Numeric (days)	≥0
Maximum_medical _improvement_reached	Whether claimant has reached maximum medical improvement	Boolean	true, false
Total_medical_costs	Total medical costs incurred (subset of total_incurred_cost)	Numeric (USD)	≥0
Total_indemnity_paid	Total indemnity/wage loss payments made	Numeric (USD)	≥0
Settlement_amount	Final settlement amount agreed upon (if claim settled)	Numeric (USD)	≥0
Reserve_adequacy	Assessment of whether reserves are adequate for expected costs	Categorical	adequate, possibly_deficient, deficient, excessive
Medical_provider_quality	Quality and appropriateness of medical care provided	Categorical	appropriate, questionable, excessive, inadequate

Critical Validation Rules

- **total_incurred_cost** ≤ ultimate_cost_prediction
- **estimated_outstanding_cost** = ultimate_cost_prediction - total_incurred_cost
- **ultimate_cost_category** must mathematically align with ultimate_cost_prediction (e.g., $65K prediction requires "50K-100K" category)
- **Settled claims**: settlement_amount = ultimate_cost_prediction

5.2.2. Ratemaking Variables

Ratemaking variables inform premium calculations and risk classification. These variables assess loss frequency and severity drivers for pricing adjustments. Category contains 8 variables.

Variable	Definition	Data Type	Valid Values
risk_level	Overall risk level classification for ratemaking and pricing decisions	Categorical	low, medium, high, extreme
causation_type	Primary mechanism/cause of the injury or incident for risk classification and experience modification	Categorical	equipment_failure, environmental_hazard, human_factor, multiple_causes
industry_risk_category	Industry risk classification for experience rating and pricing	Categorical	construction_heavy, construction_light, manufacturing_heavy, manufacturing_light, healthcare_hospital, healthcare_outpatient, transportation, retail, office_professional, agriculture, utilities, public_safety, education, other
industry_risk_level	Risk level of work environment based on industry hazards and safety factors	Categorical	low_risk, moderate_risk, high_risk, hazardous
safety_compliance	Level of safety protocol compliance observed or documented	Categorical	full, partial, none, unknown
experience_modifier_impact	Expected impact of this claim on employer's experience modification factor	Categorical	favorable, neutral, adverse, severe_adverse
employer_size_category	Size classification of employer for experience rating	Categorical	small, medium, large, enterprise
claim_frequency_indicator	Indicator of whether this is a repeat claimant or high-frequency employer	Categorical	first_claim, repeat_claimant, high_frequency_employer, unknown

Experience Modifier Impact Assessment Logic

- **Favorable**: Costs <$10K, minor injury, good safety record
- **Neutral**: Costs $10K-$50K, expected industry norm
- **Adverse**: Costs $50K-$150K, major injury, or pattern concerns
- **Severe_Adverse**: Costs >$150K, catastrophic injury, or repeated violations

5.2.3. Claims Management Variables

Claims management variables guide handling strategy, resource allocation, and settlement approach. These variables identify administrative complexity and litigation risk. Category contains 11 variables.

Variable	Definition	Data Type	Valid Values
litigation_risk	Assessed risk of claim proceeding to litigation	Categorical	low, medium, high
Settlement_likelihood	Probability of claim reaching settlement agreement vs. prolonged litigation	Categorical	high, medium, low
Management_complexity	Level of complexity in managing and handling the claim	Categorical	routine, moderate, complex, high_touch
fraud_risk	Assessed risk of fraudulent claim elements	Categorical	low, medium, high
time_to_closure_days	Estimated or actual number of days from incident to claim closure	Numeric (days)	≥0
Recommended_reserve	Recommended case reserve amount based on all available information	Numeric (USD)	≥0
work_relatedness_status	Official determination of whether injury is work-related and compensable under workers' compensation	Categorical	accepted, disputed, denied, under_investigation
return_to_work_status	Claimant's current or projected return to work status	Categorical	returned_full_duty, returned_light_duty, not_returned, permanent_disability, unknown
claim_closure_status	Current status of claim	Categorical	open_active, open_inactive, closed_settled, closed_denied, pending_settlement
Subrogation_potential	Potential for subrogation recovery from third parties	Categorical	none, low, medium, high
Claimant_cooperation_level	Level of claimant cooperation throughout claim process	Categorical	excellent, good, fair, poor, adversarial

Litigation Risk Assessment Logic

- Low: Cooperative parties, no attorney, clear liability, settlement discussions progressing
- Medium: Attorney involved, some disputes on liability/value, negotiating but slow progress
- High: Aggressive attorney, significant disputes, lawsuit threatened or filed, settlement unlikely

Settlement Likelihood Inverse Relationship:

Settlement likelihood exhibits inverse correlation with litigation risk. High litigation risk typically corresponds to low settlement likelihood and vice versa.

Cross-Category Relationships

Variables across categories exhibit interdependencies captured in Stage 2 synthesis rationales. For example:

- High injury_severity typically correlates with increased medical_complexity and elevated litigation_risk
- ultimate_cost_category drives experience_modifier_impact assessment
- work_relatedness_status affects litigation_risk and settlement_likelihood
- safety_compliance influences causation_type and industry_risk_level

The synthesis rationale documents these relationships and explains how multiple sources inform final assessments. See Section 3.3 for compound scoring mechanics that weight multi-source evidence.

5.3. JSON Schema Specifications

This section presents schema structures showing required fields, data types, and nesting relationships. Full schemas with validation rules appear in Appendix B.

5.3.1. Layer 1: Input Schema Structure

Input to Script 4 follows this structure after preprocessing by Script 2 (FHIR + document augmentation) or Script 3 (text file processing).

```
json
{
  "claim_id": "string|integer (required)",
  "patient_initials": "string (required)",
  "patient_metadata": {
    "age": "string|integer",
    "gender": "string",
    "name": "string"
  },
  "encounters": [
    {
      "encounter_id": "string (required) - UUID or encounter_NNN format",
      "document_type": "enum (required)",
      "document_text": "string (required)",
      "encounter_date": "string - ISO-8601, MM/DD/YYYY, or 'unknown'",
      "encounter_metadata": "object - FHIR metadata (Script 1/2)",
      "generation_metadata": "object - LLM generation tracking (Script 2)",
      "processing_metadata": "object - Source file tracking (Script 3)"
    }
  ],
  "processing_metadata": {
    "source_type": "enum (required) - text_file_processor|fhir_processor|document_augmentation",
    "bundle_file": "string - source filename"
    "processing_timestamp": "ISO-8601 datetime (required)",
    "processing_stats": "object - API calls, costs, errors",
    "cost_summary": "object - Token usage and costs"
  }
}
```

Valid Document Types: clinical_note, medical_provider_letter, phone_transcript, settlement_adjuster_notes, claimant_statement, adjuster_notes_initial, medical_record, treatment_notes

Validation: Schema enforces non-empty claim_id and encounters array, valid source_type enum, minimum 50 characters per document text, UTF-8 encoding. Documents failing validation are logged and excluded from processing. See Appendix B for complete validation rules.

Source Variability: Encounter metadata structure varies by source. FHIR encounters include encounter_metadata, synthetic documents include generation_metadata, text files include processing_metadata. Script 4 accommodates all formats.

5.3.2. Layer 2: Stage 1 Document-Level Schema

Stage 1 outputs follow document-type-specific schemas with variable extractions per document.

```
json
{
  "document_id": "string (required)",
  "claim_id": "string (required)",
  "document_type": "string (required)",
  "document_date": "string|null - ISO-8601 date",
  "document_weight": "number - 0.6-1.0",
  "recency_factor": "number - 0.6-1.0",
  "extracted_features": {
    "variable_name": {
```

```
      "value": "string|number|boolean|array|object",
      "confidence": "number - 0.0-1.0",
      "rationale": "string",
      "rationale_type": "string - direct|inferred|calculated",
      "source_document": "string - document_type"
    }
  },
  "narrative_analysis": "string",
  "confidence_score": "number - 0.0-1.0",
  "processing_metadata": { /* stage, model, timestamp */ }
}
```

Validation: Schema enforces required fields (document_id, claim_id, document_type, extracted_features), minimum 2 fields in extracted_features, minimum 50 characters total content. Documents failing validation are logged and excluded from processing.

Schema Specialization: Six document-type-specific function schemas define variable sets: extract_adjuster_notes_features, extract_claimant_statement_features, extract_clinical_note_features, extract_medical_provider_features, extract_phone_transcript_features, extract_settlement_notes_features. Function outputs are normalized into consistent value, confidence, and rationale structure. See Appendix B.2 for complete function schemas.

5.3.3. Layer 3: Stage 2 Claim-Level Schema

Stage 2 aggregates document-level extractions into final claim assessments organized across three actuarial categories.

```
json
{
  "claim_id": "string (required)",
  "processing_timestamp": "ISO-8601 datetime (required)",
  "actuarial_variables": {
    "reserving_variables": {
      "claim_severity": {
        "value": "enum - minor|moderate|major|catastrophic",
        "aggregate_confidence": "number - 0.0-1.0",
        "sources": [
          {
            "document_id": "string",
            "document_type": "string",
            "document_date": "string|null",
            "value": "string|number",
            "confidence": "number",
            "compound_score": "number"
          }
        ],
        "synthesis_rationale": "string",
        "conflicts_detected": "boolean",
        "resolution_method": "string|null"
      }  /* 17 reserving variables total */
    },
    "ratemaking_variables": { /* 8 variables, same structure */ },
    "claim_management_variables": { /* 11 variables, same structure */ }
  },
  "aggregated_narrative": "string (required)",
  "confidence_scores": {
    "overall": "number - 0.0-1.0 (required)",
    "reserving": "number - 0.0-1.0 (required)",
    "ratemaking": "number - 0.0-1.0 (required)",
    "claim_management": "number - 0.0-1.0 (required)",
    "data_completeness": "number - 0.0-1.0"
  }
```

}

Validation

- **Function schema** enforces **additionalProperties**: false and **maxProperties** constraints (17, 8, 11) preventing variable duplication across categories.
- **Validation detects misplaced variables** and automatically moves them to correct categories (logged in quality_flags as DUPLICATES_FIXED, MISPLACED_FIXED).
- **Cross-variable validation enforces**:
  - **total_incurred_cost** ≤ ultimate_cost_prediction (COST_PROGRESSION_ISSUE if violated),
  - **ultimate_cost_category** mathematically aligns with ultimate_cost_prediction (COST_CATEGORY_MISMATCH if misaligned),
  - **settlement_amount** = ultimate_cost_prediction for settling claims (SETTLEMENT_ULTIMATE_MISMATCH if discrepancy exceeds $100).
- **Additional quality flags** include VALIDATION_WARNING, CONSISTENCY_ISSUES_DETECTED.

Integration Contract: External systems consume this schema. Required fields (actuarial_variables, aggregated_narrative, confidence_scores) are always present. Variables within categories may be null if insufficient information. Confidence scores enable downstream systems to implement threshold-based acceptance logic based on reliability requirements. Source attribution via the sources array enables actuarial auditing. Reviewers trace each variable to supporting documents with compound scores (see Section 3.3). See Appendix B.3 for complete schema.

5.4. Metadata Structures

Metadata structures capture processing provenance, quality indicators, and confidence assessments, enabling system transparency and validation. The system implements three complementary metadata mechanisms operating at different granularities.

5.4.1. Confidence Scoring Architecture

Confidence scores operate at four hierarchical levels with distinct purposes. For confidence interpretation and generation, see Section 3.3.2.

5.4.1.1. Confidence Levels

- **Stage 1 Feature-Level:** Each extracted feature includes individual confidence (float 0-1).

  ```
  json
  {
    "primary_diagnosis": {
      "value": "right wrist fracture",
      "confidence": 0.95,
      "rationale": "..."
    }
  }
  ```

  This granular scoring enables feature-level quality assessment and selective use of extractions based on reliability thresholds.

- **Stage 1 Document-Level:** Each processed document receives aggregate confidence.

  ```
  json
  {
    "document_id": "6924817947_doc_4",
    "confidence_score": 0.85,
    "narrative_analysis": "..."
  }
  ```

Document-level scores inform compound scoring (Section 3.3) and enable document quality filtering.

- **Stage 2 Variable-Level:** Each final actuarial variable includes synthesized confidence**.**

```
json
{
  "claim_severity": {
    "value": "moderate",
    "confidence": 0.85,
    "stage1_sources": [...]
  }
}
```

Variable-level confidence enables granular evaluation of individual variables, supporting selective acceptance based on per-variable reliability rather than claim-wide thresholds.

- **Stage 2 Category-Level**: Aggregate confidence across variable categories.

```
json
{
  "confidence_scores": {
    "overall": 0.85,
    "reserving": 0.90,
    "ratemaking": 0.85,
    "claim_management": 0.88,
    "data_completeness": 1.0
  }
}
```

Category scores represent the LLM's assessed confidence in each variable category based on source quality, conflict resolution, and aggregation reliability. The LLM provides these scores as part of Stage 2 function calling output.

5.4.2. Provenance Tracking Structure

Provenance metadata creates audit trails from final variables back to source documents, enabling actuarial validation and regulatory compliance.

Stage 2 Source Attribution

Each Stage 2 variable documents all contributing Stage 1 sources with compound scoring metadata. This structure documents every source contributing to each variable, the scoring logic prioritizing sources, and the synthesis rationale explaining the final assessment. Actuaries can reconstruct decision processes and validate system outputs against original documents.

```
json
{
  "injury_severity": {
    "value": "moderate",
    "confidence": 0.90,
    "rationale": "Injury severity classified as moderate due to wrist fracture requiring intervention...",
    "rationale_type": "aggregated",
    "stage1_sources": [
      {
        "document_type": "medical_provider_letter",
        "stage1_feature": "primary_diagnosis",
        "value": "right wrist fracture",
        "confidence": 0.95
      },
      {
        "document_type": "medical_provider_letter",
        "stage1_feature": "treatment_complexity",
        "value": "surgical",
```

```
      "confidence": 0.95
    }
   ]
  }
}
```

Provenance Components:

- **stage1_sources**: List of all Stage 1 features contributing to this variable
  - **document_type**: Source document type for each contributing feature
  - **stage1_feature**: Original extracted feature name
  - **value**: Value extracted from that source
  - **confidence**: Extraction confidence from Stage 1

Rationale System

The rationale field provides human-readable synthesis logic explaining how multiple sources were aggregated into the final value. The rationale_type indicates whether the value is:

- **verbatim**: Direct extraction from single source
- **derived**: Inference from source content
- **aggregated**: Synthesis across multiple sources

Processing Metadata

Stage 1 processing creates **metadata for each processed document**, enabling cost tracking and optimization, model version auditing, and processing timeline reconstruction.

```
json
{
  "processing_metadata": {
    "api_cost": 0.001459,
    "tokens_used": {
      "prompt_tokens": 5972,
      "completion_tokens": 939,
      "total_tokens": 6911
    },
    "processing_time": 12.847,
    "model": "gpt-4o-mini-2024-07-18",
    "temperature": 0.7,
    "function_schema": "extract_clinical_note_features"
  }
}
```

When Stage 1 outputs are saved, additional **claim-level metadata** is added.

```
json
{
  "claim_id": "CLM12345",
  "stage1_features": [...],
  "processing_metadata": {
    "timestamp": "2025-11-03T22:13:47.251429",
    "documents_processed": 6,
    "stage": "stage1_feature_extraction"
  }
}
```

5.4.3. Quality Flags Structure

Stage 2 generates quality flags (string array) from three sources: confidence-based checks, non-confidence checks, and validation. Flags alert reviewers to processing concerns requiring human attention.

Confidence-Based Quality Flags Configuration

Configuration-driven system with granular thresholds for automated quality assessment. Generated by ClassProcessStage2._generate_quality_flags().

- **Configuration**

```
yaml
quality_control:
  flag_low_confidence: true          # Master switch

  # Category-level thresholds
  category_confidence_low: 0.6       # <= 0.6 = LOW
  category_confidence_medium: 0.7    # > 0.6 and <= 0.7 = MEDIUM

  # Variable-level thresholds
  variable_confidence_low: 0.5       # <= 0.5 = LOW
  variable_confidence_medium: 0.7    # > 0.5 and <= 0.7 = MEDIUM

  #Non-confidence flag controls
  flag_conflicts: true               # Enable conflict detection flags
  flag_missing_documents: true       # Enable missing document flags
  required_documents: []             # List of required document types
```

- **Flag Generation Logic**
  - **Category-Level Flags:** Check overall, reserving, ratemaking, claim_management (excludes data_completeness)

Format	Example	Trigger
{CATEGORY}-category-CONFIDENCE_LOW-{score}	RESERVING-category-CONFIDENCE_LOW-0.55	score ≤ 0.6
{CATEGORY}-category-CONFIDENCE_MEDIUM-{score}	RATEMAKING-category-CONFIDENCE_MEDIUM-0.65	0.6 < score ≤ 0.7

  - **Variable-Level Flags:** Check all individual variables within reserving_variables, ratemaking_variables, claim_management_variables

Format	Example	Trigger
{CATEGORY}-{variable_name}-CONFIDENCE_LOW-{score}	RESERVING-injury_severity-CONFIDENCE_LOW-0.45	score ≤ 0.5
{CATEGORY}-{variable_name}-CONFIDENCE_MEDIUM-{score}	CLAIM_MANAGEMENT-litigation_risk-CONFIDENCE_MEDIUM-0.65	0.5 < score ≤ 0.7

  - **Non-Confidence Flags:** Check conflicts, document coverage, missing required documents

Flag	Trigger Condition	Config Control
CONFLICTS_DETECTED	Rationale type = conflict_resolved detected	flag_conflicts: true
LIMITED_DOCUMENTS	<3 documents processed	flag_missing_documents: true
MISSING_{DOCTYPE}	Required document type missing	required_documents: [...]

  - **Validation Flags:** Generated by ClassValidateStage2.validate_stage2_output() during output validation.

Flag	Trigger Condition	Source Method
DUPLICATES_FIXED_n	Variables appeared in multiple categories (n = count)	deduplicate_and_fix_placement()
MISPLACED_FIXED_n	Variables moved to correct category (n = count)	deduplicate_and_fix_placement()

Flag	Trigger Condition	Source Method
VALIDATION_WARNING: {var}	Variable failed type/enum/range validation	definitions.validate_extraction()
CONSISTENCY_ISSUES_DETECTED	Cross-variable business logic volations	definitions.validate_stage2_consistency()
COST_CATEGORY_MISMATCH	ultimate_cost_prediction inconsistent with ultimate_cost_category	_validate_cost_category_alignment()
SETTLEMENT_ULTIMATE_MISMATCH	settlement_amount > ultimate_cost_prediction	_validate_settlement_consistency()
COST_PROGRESSION_ISSUE	total_incurred_cost > ultimate_cost_prediction	_validate_cost_progression()

- **Example Validation Scenarios:**

```
python
# COST_CATEGORY_MISMATCH example
ultimate_cost_prediction: $118,000
ultimate_cost_category: "50K-100K"        # Should be "100K-150K"
                                          → Generates: COST_CATEGORY_MISMATCH

# SETTLEMENT_ULTIMATE_MISMATCH example
settlement_amount: $150,000
ultimate_cost_prediction: $120,000        # Settlement > Ultimate
                                          → Generates: SETTLEMENT_ULTIMATE_MISMATCH

# COST_PROGRESSION_ISSUE example
total_incurred_cost: $95,000
ultimate_cost_prediction: $85,000         # Incurred > Ultimate
                                          → Generates: COST_PROGRESSION_ISSUE
```

- **Threshold Logic**
  - **LOW flag**: score <= threshold_low
  - **MEDIUM flag**: threshold_low < score <= threshold_medium
  - **No flag**: score > threshold_medium
- **Master Switch:**
  - **flag_low_confidence**: **false** disables all confidence checking
  - **Non-confidence flags** still generated if their respective settings are enabled
  - **Validation flags** always generated (cannot be disabled)
- **Operational Capabilities:** Configurable confidence thresholds and persistent quality flags create audit trails for actuarial review. Structured flag output enables downstream routing logic (auto-accept vs. manual review) in external systems.

5.5. Validation Framework

The validation framework ensures output correctness through multi-stage checks spanning schema conformance, data type validation, structural integrity, and actuarial consistency.

5.5.1. Validation Architecture

Validation occurs at five distinct stages across the pipeline.

- **Input Structure Validation (Scripts 2, 4):** Validates JSON structure from previous stages.
  - **Script 2** verifies input from Script 1 contains required fields (patient_metadata, encounters, processing_metadata) and correct source type (fhir_processor). It also validates encounter structure by ensuring each encounter contains required fields (encounter_id,

document_type, clinical_note) and that the encounters field is a non-empty list of dictionaries.
    - **Script 4** validates input from prior scripts, ensuring non-empty claim_id and encounters array. Structural failures reject inputs immediately since malformed data cannot enter processing.

- **Extraction Quality Validation (Script 4 - Stage 1): Validates extracted features at two levels.**
    - **Basic structure validation** checks for non-empty dictionary (minimum 2 fields), total content length exceeds 50 characters, and dictionary structure. Failed structure validation rejects the extraction and processing stops for that document.
    - **Individual variable validation** checks data types (string, number, integer, float, boolean, array), valid enumeration values when specified, and numeric ranges (min/max constraints). Individual variable validation failures generate warnings and quality flags (VALIDATION_WARNING) but processing continues.
- **Structural Conformance Validation (Script 4 - Stage 2):** Enforces schema constraints and corrects placement errors.
    - **JSON Schema specifications** use "additionalProperties: false" to prevent undefined variables and "maxProperties" limits (17 reserving, 8 ratemaking, 11 claim_management) to prevent schema violations.
    - **Deduplication feature** detects variables appearing in multiple categories, automatically moves misplaced variables to correct categories based on authoritative mapping, and removes duplicates keeping only correctly-placed instances. Corrections are logged with quality flags (DUPLICATES_FIXED, MISPLACED_FIXED).
- **Actuarial Consistency Validation (Script 4 - Stage 2):** Applies domain-specific business logic to detect logical inconsistencies.
    - **Cost category alignment** validates ultimate_cost_prediction matches ultimate_cost_category using predefined thresholds (<25K, 25K-50K, 50K-100K, 100K-150K, >150K). Settlement consistency checks verify settlement amounts are present for settled claims and align with ultimate cost predictions.
    - **Cost progression validation** ensures total_incurred does not exceed ultimate_cost_prediction and reserves are non-negative.
    - **Cross-variable consistency** checks relationships between variables defined in Section 5.2. Violations generate quality flags (COST_CATEGORY_MISMATCH, SETTLEMENT_ULTIMATE_MISMATCH, COST_PROGRESSION_ISSUE, CONSISTENCY_ISSUES_DETECTED) but do not block processing.
- **Document Content Validation (Script 2 - Synthetic Generation):** For synthetic data quality control only, validates master profiles exceed a configurable 800 characters with content markers and that phone transcripts include conversation markers and medical letters include clinical terminology. Validation failures stop processing.

5.5.2. Validation Methods

The **ClassVariableDefinitions** class provides validation methods used throughout processing.

- **Type Validation:** The **_validate_type** method checks extracted values match expected types using Python type checking. Supports string, number, integer, float, boolean, array, and date (ISO format string) types. Type mismatches generate warnings specifying expected versus actual type.
- **Range Validation:** The **_validate_numeric_range** method verifies numeric values fall within defined min/max constraints. Validates against validation.min_value and validation.max_value fields in variable definitions. Out-of-range values generate warnings with threshold information.

- **Enumeration Validation:** Variables with valid_values specifications are checked to ensure extracted values match allowed enumeration sets. Invalid values generate warnings listing valid options.
- **Cross-Variable Consistency:** The **validate_stage2_consistency** method applies business logic rules defined in variable definitions. Checks cross-variable relationships that must hold for outputs to be actuarially valid. Consistency failures generate warnings but allow processing to continue.

5.5.3. Error Handling Strategy

Script 4 implements rejection at pipeline entry points, flagging at aggregation.

- **Immediate Rejection**: Input validation (ClassInputOutput.load_claim_file()) rejects malformed JSON, missing required fields, or empty encounters. Stage 1 extraction validation (ClassProcessStage1._validate_extraction_result()) rejects individual document extractions failing structure or content checks. Both return None, halting processing for affected scope.
- **Quality Flag Generation**: Stage 2 validation (ClassValidateStage2.validate_stage2_output()) generates quality flags for type/enum/range violations, cross-variable inconsistencies, and cost relationship errors. Processing continues; flags attach to output.
- **Flag Propagation**: Quality flags stored at root level of Stage 2 output JSON (quality_flags field). Flags include validation warnings, deduplication reports, and consistency violations.
- **Logging**: Validation failures logged with claim ID, variable name, violated rule, and issue details. Supports debugging, audit trails, and pattern analysis

5.5.4. Integration Points

Script 4 validation occurs at three points.

- **Input Validation**: Validates JSON structure and required fields. Immediate rejection on failure. (ClassInputOutput.load_claim_file)
- **Stage 1 Validation**: Validates extraction structure and content length. Rejects failed document extraction; claim continues with remaining documents. (ClassProcessStage1._validate_extraction_result)
- **Stage 2 Validation**: Validates variable placement, data types, cross-variable consistency, and cost relationships. Generates quality flags; processing continues. (ClassValidateStage2.validate_stage2_output)

This progressive validation catches errors early, minimizes wasted processing, and ensures only validated outputs reach downstream systems. See Section 7 for script-specific validation implementation details.

Part II: Implementation Reference

6. Core Infrastructure Components

This section begins Part II of the document, transitioning from conceptual architecture to implementation reference. Each script in the pipeline implements a consistent infrastructure pattern using four core components that provide configuration management, logging, LLM interaction, and metrics tracking. This pattern ensures maintainability, consistency, and extensibility across the codebase.

6.1. Infrastructure Pattern Overview

Every script implements a consistent architectural pattern using four core infrastructure components: ClassConfig for centralized YAML configuration management, ClassLogger for structured logging with debug controls, ClassAPIClient for LLM provider abstraction with retry logic, and ClassTrackMetrics for comprehensive performance monitoring. This uniformity reduces cognitive load across scripts, enables shared infrastructure improvements, simplifies developer onboarding, and provides consistent debugging interfaces.

The pattern emerged from early prototyping that revealed code duplication across scripts for API calls, logging, and configuration parsing. Extracting these concerns into reusable components eliminated redundancy while establishing clear separation between infrastructure (execution mechanics) and business logic (processing rules). Each component maintains single responsibility and exposes clean interfaces for dependency injection.

Core Components

- **ClassConfig:** Centralized YAML Configuration Management
  - **Loads** script-specific YAML files from config/[script-name].yaml directory
  - **Validates** required configuration sections on initialization (exits with code 1 on failure)
  - **Resolves** paths consistently using Path(__file__).parent.resolve()
  - **Provides** typed accessor methods: get_api_settings(), get_logging_settings(), get_processing_settings()
  - **Supports** script-specific sections (e.g., get_medical_classification() in Script 1, get_document_types() in Script 4)
  - No environment-specific override mechanism (single config per script)
- **ClassLogger:** Structured Logging with Configurable Debug Options
  - **Console** logging with timestamps and level-based formatting
  - **JSON output** mode (json_mode=True) for web UI integration with structured events
  - **File-based** logging optional with automatic rotation by date
  - **Category-based** debug printing controlled by YAML configuration
    - **show_api_prompts**: Log LLM prompts (with character truncation via max_prompt_chars)
    - **show_api_responses**: Log LLM responses (with character truncation via max_response_chars)
    - **show_variable_definitions**: Control variable definition verbosity in prompts
    - **Stage-specific debug options** in Script 4: show_stage1_details, show_stage2_details
  - **Specialized** logging methods: info(), debug(), warning(), error(), success(), progress(), milestone(), stats()
  - **Condensed definition** logging in Script 4 via _condense_definitions_in_prompt() method
- **ClassAPIClient:** LLM Provider Abstraction with Retry Logic
  - **Provider factory pattern** via ClassCreateLLM.create_provider() supporting OpenAI and Ollama backends
  - **Exponential backoff retry logic** (configurable retry_count and retry_delay_base from YAML)

- **Comprehensive cost tracking**: per-call cost calculation using model-specific pricing rates from config
- **Token usage tracking**: separate counters for input tokens, output tokens, total tokens
- **Two primary call methods**:
  - **call_llm**(): Text completion (Scripts 1-3)
  - **call_llm_with_function**(): Structured function calling with JSON schema validation (Script 4)
- **Automatic model selection** based on call_type parameter (maps to config model definitions)
- **Cost summary generation** via get_cost_summary() or get_stats_summary()
- **Dry-run** mode support (Scripts 1, 3)

- **ClassTrackMetrics:** Performance and Cost Monitoring
  - **Timing tracking**: start_time using time.time() (Scripts 1-2) or perf_counter() (Script 4)
  - **Success/failure rate tracking** per operation (Script 4 tracks by stage and document type)
  - **Cost accumulation** with per-stage and per-document-type breakdowns (Script 4)
  - **Token usage metrics** by stage and document type (Script 4)
  - **Processing rate calculations**: items per minute, API calls per minute, tokens per dollar
  - **Method tracking**: function calling vs text fallback success rates (Script 4)
  - **Summary generation** via get_summary() and print_summary() methods
  - **JSON stats output** via output_stats() for UI consumption

Component Interactions

ClassConfig initializes first, loading all configuration settings. **ClassLogger** depends on ClassConfig for logging section settings (level, format, debug options). **ClassAPIClient** depends on both ClassConfig (for api_settings including provider, models, pricing, retry config) and ClassLogger (for logging API calls, prompts, responses). **ClassTrackMetrics** depends on ClassLogger for metric reporting via stats() method. Business logic classes receive all four components via constructor injection, enabling testability and clear dependency chains.

6.2. ClassConfig - Configuration Management

ClassConfig provides centralized YAML-based configuration management ensuring consistent parameter access across scripts. Each script has a dedicated configuration file (e.g., config/4_analysis-analyze_claims.yaml) containing all tunable parameters.

Design Philosophy: Configuration-driven design enables parameter adjustment without code changes, facilitates A/B testing of processing strategies, provides self-documenting system behavior through structured YAML, and centralizes all tunable parameters in version-controlled files.

Class Structure

```
python
class ClassConfig:
    def __init__(self, config_path: Optional[str] = None)
    def _load_config(self, config_path: Optional[str]) -> Dict

    # Common accessors (all scripts)
    def get_api_settings() -> Dict
    def get_processing_settings() -> Dict
    def get_logging_settings() -> Dict

    # Script-specific accessors
    def get_document_types() -> Dict              # Script 4
    def get_temporal_weighting() -> Dict          # Script 4
    def get_compound_scoring() -> Dict            # Script 4
    def get_stage1_settings() -> Dict             # Script 4
    def get_stage2_settings() -> Dict             # Script 4
    def get_shared_settings() -> Dict             # Script 4
```

```
def get_rationale_settings() -> Dict          # Script 4
def get_templates_dir() -> str                # Script 4
def get_medical_classification() -> Dict      # Script 1
def get_encounter_filtering() -> Dict         # Script 1
def get_cloud_storage_settings() -> Dict      # Scripts 2, 4
def get_output_structure() -> Dict            # Scripts 2, 4
def get_filename_templates() -> Dict          # Script 4
```

Configuration Sections

The configuration structure varies by script. Common sections include:

- **api_settings:** LLM provider configuration
  - Provider selection (openai or ollama)
  - Model names per stage or function (stage1, stage2, default)
  - Pricing configuration (cost per 1K tokens for input/output)
  - Retry parameters (retry_count, retry_delay_base)
  - Default token limits (default_max_tokens)
- **processing:** Processing behavior
  - Default input/output directories
  - Stage enablement flags (stage1_enabled, stage2_enabled)
  - Processing limits
- **logging:** Logging behavior
  - Log level (DEBUG, INFO, WARNING, ERROR)
  - Console and file format strings
  - File logging toggle (file_logging: true/false)
  - Debug options dictionary:
    - show_api_prompts: Display prompts sent to LLM
    - show_api_responses: Display LLM responses
    - show_stage1_details: Stage 1 debug output (Script 4)
    - show_stage2_details: Stage 2 debug output (Script 4)
    - show_variable_definitions: Full definitions in logs (Script 4)
    - max_prompt_chars: Truncation limit for prompts
    - max_response_chars: Truncation limit for responses
- **stage1_settings:** Stage 1 extraction parameters for Script 4
  - Function calling toggle (use_function_calling)
  - Template organization flags
  - Max tokens and temperature overrides
- **stage2_settings:** Stage 2 aggregation parameters for Script 4
  - Function calling toggle
  - Orchestrator configuration
  - Schema file paths
  - Max tokens and temperature settings
- **shared_settings:** Cross-stage configuration for Script 4
  - Templates directory path
  - Rationale system configuration (mode, enabled/disabled, selective fields)
- **temporal_weighting:** Temporal decay parameters for Script 4
  - Enable flag (enabled: true/false)
  - Decay function type (stepped, exponential, linear)
  - Stepped decay thresholds (days and factor pairs)
  - Missing date handling strategy and default factor

    - Document-specific overrides
- **compound_scoring:** Source prioritization for Script 4
    - Enable flag and component toggles
    - Default values for missing scores
    - Minimum score thresholds
    - Score presentation options
- **document_types:** Document type definitions
    - Per-type model parameters (max_tokens, temperature)
    - Priority levels (integer, lower = higher priority)
    - Weight values (0.0-1.0, higher = more authoritative)
    - Template filenames and schema paths

Path Resolution

ClassConfig resolves paths relative to the **script directory**, not project root, supporting both absolute paths and relative paths. Path resolution logic in _load_config():

```
python
if config_path is None:
    config_file = self.script_dir / "config" / "[script-name].yaml"
else:
    config_path_obj = Path(config_path)
    config_file = config_path_obj if config_path_obj.is_absolute() else self.script_dir / config_path
```

Validation

Configuration loading performs validation on required sections and exits immediately on failure with generic exit code, sys.exit(1).

- **Required sections check:** Verifies presence of mandatory sections exits immediately on failure:
    - Script 1: ['api_settings', 'processing', 'medical_classification']
    - Script 4: ['api_settings', 'processing', 'logging', 'document_types', 'shared_settings']
- **No deep validation** such as data types, enum values, or file path accessibility beyond config file existence.

Example Configuration Snippet:

```
yaml
api_settings:
  provider: "openai"
  default_max_tokens: 2000
  retry_count: 1
  retry_delay_base: 2
  openai:
    models:
      stage1: "gpt-4o-mini-2024-07-18"
      stage2: "gpt-4o-mini-2024-07-18"
    pricing:
      gpt_4o_mini_input_cost_per_1k: 0.00015
      gpt_4o_mini_output_cost_per_1k: 0.0006

logging:
  level: DEBUG
  console_format: "%(asctime)s %(levelname)-5s : %(message)s"
  file_logging: false
  debug_options:
    show_api_prompts: true
    show_stage1_details: true
    max_prompt_chars: 2000
```

```
processing:
  default_input_dir: "output/2_data-add_documents"
  default_output_dir: "output/4_analysis-analyze_claims"
  stage1_enabled: true
  stage2_enabled: true

stage1_settings:
  use_function_calling: true
  max_tokens: 2000
  temperature: 0.3

temporal_weighting:
  enabled: true
  decay_function: "stepped"
  stepped_decay:
    thresholds:
      - days: 30
        factor: 1.0
      - days: 90
        factor: 0.9
  missing_date_factor: 0.7

compound_scoring:
  enabled: true
  use_feature_importance: true
  missing_importance_default: 0.5

document_types:
  medical_provider_letter:
    priority: 1
    weight: 1.0
    max_tokens: 2000
    temperature: 0.2
    specialized_template: "template_medical_provider.py"
    function_schema_file: "stage1_medical_provider.json"

# See Appendix A for complete configuration reference
```

6.3. ClassLogger - Logging System

ClassLogger provides structured logging with category-based debug controls and JSON output mode for web UI integration. The logger balances information visibility (detailed logs for debugging) with output cleanliness (concise logs for production).

Logging Capabilities

- **Severity Levels:** Standard levels (DEBUG, INFO, WARNING, ERROR, CRITICAL) with emoji-based visual distinction in console output
- **Timestamp Precision:** Microsecond-precision timestamps for performance analysis
- **Category-Based Debug:** Enable/disable debug output per category without changing log level
- **JSON Output Mode:** Structured JSON events for programmatic consumption by web UI
- **File Logging:** Uses FileHandler with date-stamped filenames.

Debug Categories: ClassLogger supports granular debug control through YAML configuration.

- **show_api_prompts**: Log LLM prompts with character truncation (max_prompt_chars)
- **show_api_responses**: Log LLM responses with character truncation (max_response_chars)
- **show_variable_definitions**: Control variable definition verbosity in debug log output
- **show_stage1_details**: Log Stage 1 processing details (Script 4)
- **show_stage2_details**: Log Stage 2 processing details (Script 4)
- **show_file_processing**: Log file processing progress (Script 3)

- **show_json_creation**: Log JSON structure creation (Script 3)
- **show_encounter_details**: Log encounter extraction details (Script 1)

Each category can be independently enabled/disabled via configuration, allowing targeted debugging without overwhelming output.

Class Methods:

```
python
class ClassLogger:
    def __init__(self, config: ClassConfig, json_mode: bool = False)

    # Standard logging
    def info(self, message: str)
    def warning(self, message: str)
    def error(self, message: str)
    def debug(self, message: str)
    def success(self, message: str)

    # Debug logging
    def debug_api_prompt(self, prompt: str, call_type: str)
    def debug_api_response(self, response: str, call_type: str)
    def debug_stage1(self, message: str)  # Script 4 only
    def debug_stage2(self, message: str)  # Script 4 only

    # Structured events (JSON mode compatible)
    def progress(self, message: str, current: int = None, total: int = None, **kwargs)
    def milestone(self, message: str, status: str = 'info', **kwargs)
    def stats(self, data: Dict)
```

JSON Output Mode

When json_mode=True, logger outputs structured JSON events to stdout via print() while simultaneously continuing standard logging through the configured console handler. This means both JSON events and text logs are emitted, enabling dual consumption (web UI parsing JSON from stdout, human debugging from console handler stream).

```
json
{
  "timestamp": "2024-11-02T14:32:15.123456",
  "event_type": "progress",
  "message": "Processing document",
  "data": {
    "current": 3,
    "total": 7,
    "stage": "stage_1"
  }
}
```

Usage Pattern:

```
python
# Initialize logger with config
logger = ClassLogger(config, json_mode=False)

# Standard logging
logger.info("Processing claim CLM12345")
logger.warning("Low confidence score: 0.58")
logger.error("API call failed after 3 retries")
logger.success("Processing complete")

# Debug logging (controlled by YAML config)
logger.debug_api_prompt(prompt, call_type="stage1_extraction")
logger.debug_api_response(response, call_type="stage1_extraction")
```

```
# Structured events for UI integration
logger.progress("Processing documents", current=3, total=7, stage="stage_1")
logger.milestone("Stage 1 complete", status="success", documents_processed=7)
logger.stats({'total_cost': 0.05, 'total_tokens': 1234})
```

6.4. ClassAPIClient - LLM Provider Abstraction

ClassAPIClient abstracts LLM API interaction behind a provider-agnostic interface, enabling multiple backend support (OpenAI, Ollama, future providers) through a factory pattern. This design decouples business logic from provider-specific API details.

Provider Abstraction Philosophy

The abstraction layer provides single integration point for all LLM calls, consistent error handling across providers, centralized cost tracking, simplified retry logic, and unified function calling interface despite API differences.

Core Design Pattern

Abstract base class ClassLLM defines provider interface implemented by ClassLLMOpenAI and ClassLLMOllama. ClassCreateLLM factory instantiates providers based on configuration.

```
python
class ClassLLM(ABC):
    """Abstract base class defining provider interface"""
    @abstractmethod
    def call_llm(self, messages, model, max_tokens, temperature, call_type) -> Dict:

    @abstractmethod
    def call_llm_with_function(self, messages, schema, function_name,
                               model, max_tokens, temperature, call_type) -> Dict:
    @abstractmethod
    def get_cost_summary(self) -> Dict

class ClassLLMOpenAI(ClassLLM):
    # Implements interface using openai library

class ClassLLMOllama(ClassLLM):
    # Implements interface using ollama library

class ClassCreateLLM:
    """Factory for creating LLM providers"""

    @staticmethod
    def create_provider(config, logger, dry_run) -> ClassLLM:
        provider_name = config.get('provider', 'openai')

        if provider_name == 'openai':
            provider_config = config.get('openai', {})
            return ClassLLMOpenAI(config, logger, provider_config, dry_run)
        elif provider_name == 'ollama':
            provider_config = config.get('ollama', {})
            return ClassLLMOllama(config, logger, provider_config, dry_run)
        else:
            raise ValueError(f"Unknown provider: {provider_name}")
```

Supported Providers

The system supports OpenAI as the primary production LLM provider, utilizing the gpt-4o and gpt-4o-mini models. This provider supports native function calling and API-provided cost tracking. Ollama integration is limited to preliminary testing with Script 1; further validation is required across all other scripts.

Configuration Structure

```yaml
# OpenAI Configuration
api_settings:
  provider: openai
  openai:
    api_key_env: OPENAI_API_KEY
    models:
      default: gpt-4o-mini-2024-07-18
      classification: gpt-4o-mini-2024-07-18
      clinical_notes: gpt-4o-2024-08-06
    pricing:
      gpt_4o_mini_input_cost_per_1k: 0.00015
      gpt_4o_mini_output_cost_per_1k: 0.0006
  retry_count: 3
  retry_delay_base: 2
  default_max_tokens: 1500

# Ollama Configuration
api_settings:
  provider: ollama
  ollama:
    base_url: http://localhost:11434/v1
    models:
      default: mistral
  retry_count: 3
  retry_delay_base: 2
```

Retry Logic with Exponential Backoff

ClassAPIClient implements robust configurable retry logic for transient failures:

- **Max Retries**: 3 (default)
- **Base Delay**: 2 seconds (default)
- **Backoff Formula**: delay = base_delay ** (attempt + 1) produces 2s, 4s, 8s
- **Retry on All Exceptions**: Implementation retries on any exception without filtering by type

Cost and Token Tracking

ClassAPIClient tracks cumulative metrics in ClassAPIClient wrapper.

- **total_cost**: USD accumulator based on provider pricing
- **total_calls**: API call counter
- **total_tokens**: Combined input + output tokens
- **total_input_tokens**: Prompt tokens
- **total_output_tokens**: Completion tokens

ClassLLMOpenAI tracks metrics internally. ClassLLMOllama tracks tokens when usage data available but always reports zero cost. **get_cost_summary**() returns:

```python
{
  'total_cost': float,
  'total_calls': int,
  'average_cost_per_call': float,
  'total_tokens': int,
  'total_input_tokens': int,
  'total_output_tokens': int,
  'total_time': float,
  'average_time_per_call': float
}
```

Token tracking enables cost projection ("processing 1000 claims at current rate costs $X") and budget monitoring (stop processing if approaching budget limit). Cost summary available via get_cost_summary() method for inclusion in processing reports.

Function Calling Implementation

ClassAPIClient provides **call_llm_with_function**() method leveraging function calling to enforce JSON schema compliance, eliminating JSON parsing errors from malformed LLM outputs. See Section 3.1 for Stage 1 extraction implementation using function calling.

```
python
response = self.client.chat.completions.create(
    model=model,
    messages=messages,
    functions=[schema],
    function_call={"name": function_name},
    max_tokens=max_tokens,
    temperature=temperature
)
```

ClassAPIClient Interface

ClassAPIClient maintains backward compatibility by converting simple prompt strings to messages format and wrapping provider calls.

```
python
class ClassAPIClient:
    def __init__(self, config: ClassConfig, logger: ClassLogger, dry_run: bool = False)
    def call_llm(self, prompt, model, max_tokens, temperature, call_type) -> Dict
    def _calculate_input_cost(self, tokens: int, model: str) -> float
    def _calculate_output_cost(self, tokens: int, model: str) -> float
    def get_cost_summary(self) -> Dict  # Delegates to provider
```

6.5. ClassTrackMetrics - Metrics Collection

ClassTrackMetrics aggregates processing statistics for performance analysis, cost monitoring, and quality assessment. Metrics inform system optimization and provide quantitative data for actuarial validation.

Implementation Differences by Script

- **Script 1 (FHIR Processing)**
  - **Tracks**: files processed, encounters processed, API calls, tokens, cost
  - **Methods**: start_file_processing(), end_file_processing(), record_api_call(), get_processing_rate(), get_api_rate(), get_efficiency(), output_stats()
  - **Metrics**: encounters/minute, API calls/minute, tokens/dollar, average cost/file
- **Script 4 (Claims Analysis):**
  - **Tracks**: Stage 1/Stage 2 success/failure, cost by stage and document type, token usage by stage and document type, processing times, method usage (function calling vs text fallback)
  - **Methods**: record_api_call(), record_token_usage(), comprehensive metrics with stage and document type breakdowns
  - **Additional tracking**: generation success rates, method-specific statistics, detailed cost attribution

Core Tracked Metrics (Script 4)

- **Processing Counters**
  - Files processed/successful/failed
  - Claims processed
  - Stage 1: documents processed/successful/failed

    - Stage 2: aggregations processed/successful/failed
- **Cost Metrics**
    - Total cost
    - Cost by stage (stage1, stage2)
    - Cost by document type
    - Cost by method (function_calling, text_fallback)
- **Token Usage**
    - Total input/output/total tokens
    - By stage: input/output/total for stage1, stage2
    - By document type: input/output/total per document type
- **Performance Timing**
    - Processing times per operation
    - Stage processing times (list of durations for stage1, stage2)
    - Average time calculations
- **Quality Metrics**
    - Generation success rates by stage
    - Method usage statistics (success count, total attempts, cost per method)

Class Interface (Script 4)

```
python
class ClassTrackMetrics:
    def __init__(self, logger: ClassLogger)
    def record_api_call(self, api_response, stage, document_type, method, success)
    def record_token_usage(self, usage: Dict, context: str, doc_type: str = None)
    def get_summary(self) -> Dict
    def print_summary(self)
```

Usage Pattern

```
python
metrics = ClassTrackMetrics(logger)

# Track Stage 1 document processing
start_time = time.time()
try:
    result = process_stage1_document(doc, doc_type)
    api_response = result['api_response']
    metrics.record_api_call(api_response, stage="stage1", document_type=doc_type, success=True)
except Exception as e:
    metrics.record_api_call(error_response, stage="stage1", document_type=doc_type, success=False)

# Track Stage 2 aggregation
result = aggregate_stage2(features)
metrics.record_api_call(result['api_response'], stage="stage2", success=True)

# Output summary
metrics.print_summary()
```

Summary Output Structure

Metrics summary includes:

- File and claim processing counts with success/failure breakdown
- Total cost with per-claim average
- Cost breakdown by stage (stage1, stage2) and document type
- Token usage totals and averages
- Processing time statistics
- Success rates by stage

- Method usage statistics (function calling vs fallback)

This summary provides at-a-glance assessment of processing run quality and cost efficiency, informing decisions about production readiness and configuration tuning.

6.6. Variable Definitions System

The variable definitions system manages actuarial variable specifications, providing structured definitions for LLM prompts and enabling feature importance mapping for compound scoring.

6.6.1. System Architecture

Class Interface

```
python
class ClassVariableDefinitions:
    def __init__(self, definitions_path)

    # Stage 1: Document-specific variable access
    def get_stage1_variables_for_document(self, document_type, include_secondary) -> List[str]
    def get_stage1_definition(self, variable_name) -> Optional[Dict]
    def format_stage1_definition(self, variable_name, detail_level) -> str
    def build_stage1_definitions_section(self, document_type, mode) -> str

    # Stage 2: Feature importance access
    def get_feature_importance(self, variable_name) -> Dict[str, float]
    def get_stage2_definition(self, variable_name) -> Optional[Dict]
```

- **Configuration Source:** module_variable_definitions.yaml
- **Variable Types:**
  - **Stage 1:** Document-specific extraction features (62 variables)
  - **Stage 2:** Final actuarial variables for reserving, ratemaking, claim management (33 variables, 15 with complete feature_importance)

6.6.1.1. Variable Definition Structure

Stage 2 variables include **feature_importance** mappings

```
yaml
# definitions/module_variable_definitions.yaml
stage2_variables:
  claim_severity:
    category: "RESERVING"
    description: "Overall claim severity classification"
    data_type: "string"
    enum: ["minor", "moderate", "major", "catastrophic"]
    feature_importance:
      settlement_adjuster_notes.claim_severity: 0.98
      settlement_adjuster_notes.settlement_amount: 0.95
      adjuster_notes_initial.initial_severity: 0.50
```

6.6.1.2. Document-Variable Mapping Structure

Primary variables are document strengths. **Secondary variables** provide context enrichment

```
yaml
document_variable_mapping:
  phone_transcript:
    primary_variables:
      - incident_type
      - injury_type_and_body_part
      - claimant_cooperation_level
    secondary_variables:
      - family_financial_pressure
      - credibility_assessment
```

```
medical_provider_letter:
  primary_variables:
    - primary_diagnosis
    - treatment_complexity
    - disability_rating
```

6.6.1.3. Feature Importance Mapping

Feature importance specifies document type authority for each Stage 2 variable. Weights (0.0-1.0) reflect source credibility and completeness. Variables without feature_importance mappings for a document type use configurable default score of 0.5.

- **Format**: "document_type.stage1_variable_name": importance_weight
  - **Document-Variable Mapping** links document types to relevant Stage 1 variables.
- **Weight Ranges by Source Authority**
  - 0.95-1.0: Settlement documents with complete claim history
  - 0.85-0.95: Medical professionals with clinical expertise
  - 0.75-0.85: Experienced adjusters with investigation findings
  - 0.50-0.70: Initial assessments with limited information
  - 0.30-0.50: Self-reported data requiring verification

6.6.1.4. Compound Scoring Integration

During Stage 2 aggregation, feature importance adjusts confidence scores:

compound_score = document_weight × confidence × recency × feature_importance

6.6.2. Processing Logic

Each variable definition provides LLM-ready specifications embedded in extraction prompts, ensuring consistent extraction criteria across documents.

- **Variable Loading by ClassVariableDefinitions**
  - Parse YAML configuration
  - Validate structure
  - Index by category (stage1_variables, stage2_variables)
  - Load document-variable mappings
- **Stage 1 Prompt Generation**

  When generating Stage 1 prompts, variable definitions are embedded providing clear extraction criteria.

```
python
def generate_stage1_prompt(document_text, document_type):
    # Get document-specific variables
    var_defs = ClassVariableDefinitions()
    relevant_vars = var_defs.get_stage1_variables_for_document(document_type)

    # Build definitions section with tiered detail levels
    definitions_section = var_defs.build_stage1_definitions_section(
        document_type, mode="tiered"
    )

    prompt = f"{definitions_section}\n\nDocument Text:\n{document_text}"
    return prompt
```

- **Stage 2 Feature Importance Lookup**

During Stage 2 compound scoring, feature importance weights are retrieved based on variable and source document type.

```
python
def calculate_compound_score(variable_name, document_type, stage1_var, base_confidence):
    var_defs = ClassVariableDefinitions()
    importance_map = var_defs.get_feature_importance(variable_name)

    # Lookup: "document_type.stage1_variable_name"
    key = f"{document_type}.{stage1_var}"
    importance = importance_map.get(key, 0.5)  # Default if not defined

    compound_score = (document_weight * base_confidence *
                      recency_factor * importance)
    return compound_score
```

6.6.2.1. Tiered Detail Levels (Stage 1)

Tiered approach optimizes token usage while ensuring critical variables receive comprehensive guidance.

- **Comprehensive (Tier 1 - Critical Variables):**
  - Full description, data type, unit
  - Complete extraction rules
  - Valid values enumeration
  - Calculation methods
  - Validation rules (min/max, rationale requirements)
  - Multiple examples with correct/incorrect outputs
  - Critical rules and consistency checks
- **Standard (Tier 2 - Important Variables):**
  - Description, data type
  - Extraction rules
  - Valid values
  - Single example with output
- **Minimal (Tier 3 - Basic Variables):**
  - Description
  - Data type only

6.6.2.2. Integration with Compound Scoring

Feature importance enables document-specific source weighting. Medical provider letters receive high weights for clinical variables, settlement notes receive high weights for cost variables, initial adjuster notes receive lower weights reflecting limited information at claim opening.

This granular control improves aggregation accuracy by recognizing that different document types have varying authority levels for different variables.

6.7. Error Handling Strategy

The system implements multi-level error handling balancing robustness (continue processing despite errors) with safety (stop processing for critical failures).

Error Levels

Level	Description	Action	Example
Recoverable	Non-critical errors affecting single document/claim	Log warning, flag item, continue processing	Validation warning, extraction structure failure
Severe	Critical errors affecting processing capability	Log error, retry with backoff, halt if unrecoverable	API timeout, connection failure

Level	Description	Action	Example
Fatal	Unrecoverable errors preventing processing	Log error, stop execution	Missing config file, invalid configuration

Error Categorization

Errors are categorized by source and impact.

- **Input Errors**: Invalid JSON, missing required fields, empty encounters → Return None, skip processing
- **API Errors**: Timeouts, connection failures → Retry with exponential backoff (see Section 6.4)
- **Processing Errors**: Extraction failures, validation warnings → Log context, continue with quality flags
- **System Errors**: Missing configuration, invalid YAML → sys.exit(1), processing cannot proceed

Recovery Mechanisms

- **Retry with Backoff**: API failures retry with exponential backoff (default: 3 retries, delays 2s → 4s → 8s). See Section 6.4 for implementation details.
- **Graceful Degradation**: Processing continues with partial results when individual document extractions fail. Failed documents logged; claim continues with remaining documents.
- **Detailed Logging**: All errors log full context (claim ID, document type, file path, error message) for diagnosis.

Error Handling Pattern

- **Input Validation** (returns None on failure):

```
python
def load_claim_file(self, file_path: Path) -> Optional[Dict]:
    try:
        with open(file_path, 'r') as f:
            data = json.load(f)

        if 'claim_id' not in data:
            self.logger.error(f"Missing claim_id in {file_path.name}")
            return None

        if 'encounters' not in data or not data['encounters']:
            self.logger.error(f"No encounters found in {file_path.name}")
            return None

        return data

    except json.JSONDecodeError as e:
        self.logger.error(f"Invalid JSON in {file_path.name}: {e}")
        return None
```

- **Extraction Validation** (returns False on failure):

```
python
def _validate_extraction_result(self, data: Dict, document_type: str) -> bool:
    if not data or not isinstance(data, dict):
        self.logger.error(f"Invalid data structure for {document_type}")
        return False

    if len(data) < 2:
        self.logger.error(f"Insufficient data in {document_type}: only {len(data)} fields")
        return False

    return True
```

- **API Retry** (raises exception after exhausting retries):

```
python
for attempt in range(self.max_retries):
    try:
        result = self.provider.call_llm(...)
        return result
    except Exception as e:
        self.logger.error(f"API call attempt {attempt + 1}/{self.max_retries} failed: {e}")
        if attempt < self.max_retries - 1:
            delay = self.base_delay ** (attempt + 1)
            time.sleep(delay)
        else:
            raise
```

- **Configuration Errors** (fatal, exits immediately):

```
python
if not config_file.exists():
    print(f" Config file not found at: {config_file}")
    sys.exit(1)

if section not in config:
    print(f" Missing required config section: {section}")
    sys.exit(1)
```

This pattern ensures most errors don't halt processing while critical failures get appropriate attention. The "flag, don't fail" philosophy (see Section 5.5.3) applies throughout. Questionable results are flagged for human review rather than silently discarded or blocking processing.

Operator Visibility

All errors are logged with sufficient context for diagnosis:

- Input errors: File path, error type (JSON decode, missing field)
- Extraction errors: Document type, claim ID, validation failure reason
- API errors: Attempt number, error message, retry delay
- Configuration errors: Missing file/section, YAML parse error

JSON mode enables web UI to display processing status and error summaries. Metrics tracking (Section 6.5) provides processing rate monitoring.

7. Script Implementation Details

This section provides implementation specifications for each of the four scripts comprising the processing pipeline. Scripts 1-2 generate synthetic test data for system validation with known ground truth. Scripts 3-4 form the production analysis pipeline processing real claims documents. Each script leverages core infrastructure components while implementing script-specific processing logic

7.1. Script 1: FHIR Bundle Processor

Script 1 converts FHIR medical bundles into unified JSON format with LLM-generated clinical summaries. This script enables the synthetic data workflow, providing test cases with complete medical histories and known outcomes for system validation. The script processes Synthea-generated FHIR bundles and standard FHIR R4 bundles containing encounters with diagnoses, procedures, medications, and observations. These structured elements are used to generate narrative clinical notes but are not preserved as separate structured fields in the output.

Architecture Components

- **ClassProcessFhir**: Main orchestrator coordinating FHIR parsing, encounter filtering, summarization.
- **ClassProcessEncounter**: Filters encounters by recency and prioritizes based on date, configurable to limit encounters per patient or focus on recent time periods..
- **ClassSummarizeClinicalData**: Generates narrative clinical notes via LLM from structured FHIR data.

Processing Flow

- **Load FHIR Bundle**: Parse JSON file, detect bundle type (standard FHIR R4 or Synthea native format), validate structure.
- **Extract Patient Information**: Parse Patient resource for name, age, gender, and generate patient initials.
- **Process Encounters**: Apply filtering rules (recency, max count), then iterate through filtered encounters. If max encounter limit is configured, encounters are prioritized by most recent date first.
- **Filter and Classify Encounters**: Apply pre-processing filters (recency, max count) via ClassProcessEncounter. Classify each encounter's relevance using LLM-based analysis across categories (high_cost, moderate_cost, low_cost, preventive, administrative). Skip encounters classified as not relevant to reduce processing of low-value encounters.
- **Extract Clinical Data for Prompt Generation:** For each relevant encounter, extract conditions, procedures, medications, observations, and other clinical elements from FHIR structures. This data is formatted into a prompt for LLM-based clinical note generation. The structured data is not preserved in the output; instead, it is transformed into narrative clinical notes.
- **Generate Clinical Note Summary**: Construct prompt with structured clinical data, call LLM to generate narrative clinical note, simulate realistic medical documentation style.
- **Output Unified JSON**: Structure all encounters into standardized format and write to output directory.

Key Features

- **FHIR R4 Compliance:** Full support for FHIR R4 and Synthea-generated FHIR bundles.
- **Token Optimization:** Summarizes only relevant clinical data, filters out verbose diagnostic details unnecessary for claims analysis, significantly reduces LLM token consumption.

- **Encounter Relevance Classification:** LLM-based classification criteria (high/moderate/low cost, preventive, administrative) are embedded in the classification prompt. Configurable aspects include recency filtering, max encounters per patient, and model selection.
- **Medical Terminology in Narratives:** Extracts medical term display names from FHIR code structures (ICD-10, CPT, SNOMED) and includes them in prompts for LLM narrative generation. The structured codes themselves are not preserved in the output; clinical terminology appears only within generated narrative text.
- **Metadata Preservation:** Retains encounter ID, date, type, and source format. Processing metadata includes classification results, timestamps, and models used. Detailed clinical context (reasons, locations, providers) is incorporated into narrative text but not preserved as separate structured fields.
- **Cost Tracking:** Per-encounter cost calculation, cumulative cost monitoring.

Output Schema:

```
json
{
  "claim_id": "string - Unique claim identifier (UUID)",
  "patient_initials": "string - e.g., JD",
  "patient_metadata": {
    "age": "integer",
    "gender": "string",
    "name": "string"
  },
  "encounters": [
    {
      "encounter_id": "string - FHIR resource ID",
      "document_type": "enum - [emergency, inpatient, outpatient, ambulatory]",
      "document_text": "string - LLM-generated narrative summary",
      "encounter_date": "ISO-8601 date",
      "document_metadata": {
        "generated_by": "fhir_processor",
        "   models_used": {
          "classification_model": "string",
          "clinical_note_model": "string"
        }
        "processing_timestamp": "ISO-8601 datetime"
      }
    }
  ],
  "processing_metadata": {
    "source_type": "fhir_processor",
    "bundle_file": "string",
    "relevant_encounters": "integer",
    "total_encounters_evaluated": "integer",
  }
}
```

This output format is compatible with Script 2 (document augmentation) and Script 4 (claims analysis), enabling seamless pipeline integration.

7.2. Script 2: Document Augmentation Engine

Script 2 enhances FHIR-derived JSON with synthetic documents (adjuster notes, phone transcripts, settlement documents, claimant statements), creating complete claim ecosystems for testing. FHIR bundles contain only medical data; real claims include investigation notes, communication records, and settlement documentation. Script 2 generates these additional document types while maintaining consistency with clinical context.

Architecture Components

- **ClassDocumentSchemas**: LLM function calling schemas enforcing document structure for each document type
- **ClassExtractFeatures**: Extracts key facts from FHIR encounters for document generation context
- **ClassGenerateProfile**: Creates master profile summarizing claim context for document generation
- **ClassGenerateDocuments**: Main engine coordinating document generation across types
- **ClassValidateInput**: Validates Script 1 output structure before augmentation
- **ClassProcessDocuments**: Orchestrates batch processing, file I/O, error handling

Document Generation Process

- **Read Script 1 Output**: Load JSON file produced by FHIR processor
- **Validate Input Structure**: Check source_type: "fhir_processor" confirming Script 1 origin, verify required fields, validate encounter array non-empty
- **Extract Clinical Features**: Extract key facts from using ClassExtractFeatures
- **Generate Master Profile**: Create comprehensive claim summary using ClassGenerateProfile with function calling schema, consolidate patient demographics, injury details, treatment timeline, outcome information
- **Generate Documents by Type**: For each configured document type (phone_transcript, adjuster_notes_initial, medical_provider_letter, settlement_adjuster_notes, claimant_statement):
  - Load document-specific function calling schema from ClassDocumentSchemas
  - Build prompt combining master profile with document-type-specific instructions
  - Call LLM with function calling to generate structured document
  - Validate generated document structure
  - Add document metadata (document_id, document_type, priority, timestamp, source)
  - Append to documents array
- **Merge with Original Data**: Combine generated documents with original encounters, preserve all Script 1 metadata
- **Update Metadata**: Add augmentation metadata (documents_generated, document_types_added, augmentation_timestamp)
- **Output Enhanced JSON**: Write complete JSON with medical + non-medical documents

Document Types Generated

Document Type	Description	Priority	Key Content
phone_transcript	Initial injury report call	HIGH	First-person incident account, conversational dialogue, injury description, next steps
adjuster_notes_initial	Investigation documentation	HIGH	Liability assessment, workplace safety observations, initial actions
medical_provider_letter	Physician correspondence	MEDIUM	Clinical summary, treatment plan, prognosis, work restrictions
settlement_adjuster_notes	Settlement evaluation	HIGH	Expense calculation, wage loss analysis, settlement recommendation
claimant_statement	First-person narrative	MEDIUM	Incident description, pain levels, functional limitations, daily life impact

Generation Features

- **Function Calling Schemas**: Enforce structured output for each document type ensuring consistent format
- **Master Profile Strategy**: Single comprehensive claim summary shared across all document generations for consistency

- **Contextual Consistency**: Generated documents align with injury type, severity, treatment duration from FHIR encounters
- **Priority Assignment**: Priorities based on actuarial relevance for downstream analysis
- **Temperature Control**: Document-specific temperature settings from config (higher for narrative documents, lower for structured notes)
- **Token Management**: Configurable max_tokens per document type

Output Enhancement: Script 2 extends Script 1 format by adding documents array while preserving all original content. Output is compatible with Script 4 for two-stage analysis.

7.3. Script 3: Text Preprocessor

Script 3 converts raw unstructured text files into structured JSON format compatible with Script 4 analysis. This script handles real-world claims data without LLM calls, performing pure text processing for efficient preprocessing of large document volumes. The script reads text files and structures everything into the unified JSON format Script 4 expects.

Architecture Components

- **ClassProcessTextDocuments:** Main processing engine coordinating:
  - Patient folder iteration
  - Text file reading and grouping
  - Metadata extraction from YAML files
  - JSON structure construction
  - Output file generation

Processing Pipeline

1. Directory Scanning

- Scan configured input directory for patient folders
- Each subfolder represents one patient
- Identify {patient_initials}_metadata.txt files (YAML format)
- Locate text files matching pattern: {patient_initials}_{claim_id}_{document_type}.txt

2. Metadata Loading

- Parse YAML metadata file per patient folder
- Required fields: patient_initials, age, gender, name
- Validate metadata completeness before processing

3. File Grouping

- Parse filenames to extract: patient_initials, claim_id, document_type
- Patient_initials from filename used for grouping only; authoritative patient data comes from metadata YAML file
- Group files by claim_id (multiple claims per patient supported)
- Group documents by document_type within each claim
- Handle duplicate document types via merge logic (concatenate content with separators)

4. Text Processing

- Read files with UTF-8 encoding (configurable)
- Strip whitespace via .strip()
- Validate minimum content length (configurable, default: 50 characters)
- Extract document dates from content
- Preserve original text without transformation

5. Date Extraction

- Apply regex patterns to full document content
- Return first match found; if no match, return "unknown"
- No validation for future dates or ambiguous dates (processing continues regardless)
- Extracted date stored as string in encounter object

6. Encounter Construction

- Create encounter objects for each document
- Assign sequential encounter_id
- Include source file tracking and processing metadata

7. JSON Output

- Group all documents per claim into single JSON file
- Filename pattern: {patient_initials}_{claim_id}.json
- Include patient metadata from YAML file
- Add processing_metadata
- Patient_initials from filename used for output filename; patient_initials from metadata YAML included in JSON content

Output Format

```
json
{
  "claim_id": "string - e.g., '9124018720'",
  "patient_initials": "string - lowercase, from filename parsing, e.g., 'pm'",
  "patient_metadata": {
    "age": "integer",
    "gender": "string",
    "name": "string",
    "patient_initials": "string - from metadata YAML",
    "occupation": "string - optional field from metadata"
  },
  "encounters": [
    {
      "encounter_id": "string - sequential format: 'encounter_001', 'encounter_002'...",
      "document_type": "string - standardized type from mapping (e.g., 'adjuster_notes_initial')",
      "document_text": "string - complete document content with original formatting preserved",
      "encounter_date": "string - extracted date in original format or 'unknown' if not found",
      "processing_metadata": {
        "source_files": ["array - single filename or multiple if merged"],
        "processing_timestamp": "ISO-8601 datetime with microseconds",
        "processor": "text_file_processor"
      }
    }
  ],
  "processing_metadata": {
    "source_type": "text_file_processor",
    "processing_timestamp": "ISO-8601 datetime with microseconds",
    "patient_folder": "string - source folder name (e.g., 'Patient_XY')",
    "encounters_processed": "integer - count of encounter objects created",
    "text_files_combined": "integer - count of text files processed"
  }
}
```

Document Type Mapping

The script maps various filename conventions to standardized types.

- phone_transcript, phone, transcript → phone_transcript
- adjuster_notes_initial, adjuster, adjuster_initial → adjuster_notes_initial
- medical_provider_letter, medical, provider → medical_provider_letter
- settlement_adjuster_notes, settlement → settlement_adjuster_notes

- claimant_statement, claimant, statement → claimant_statement
- clinical_note, clinical, note → clinical_note

Quality Control

- **Validation Checks**
  - **Minimum character count:** Validates total characters after .strip() against configurable threshold (default: 50 characters total)
  - **Valid filename format:** Requires minimum 2 underscore-separated parts (patient_initials_claim_id); additional parts joined as document_type
  - **Metadata file presence:** Requires *_metadata.txt YAML file in patient folder; validates presence of required fields (patient_initials, age, gender, name)
  - **File accessibility:** Standard Python file reading with UTF-8 encoding; failures caught in exception handling
- **Error Handling**
  - **continue_on_file_error**: True (default config): Individual file errors logged and skipped; processing continues with remaining files
  - **save_partial_results**: True (default config): On catastrophic failure, returns partial results dict with error information
  - **Skipped files tracked** in folder_result["skipped_files"] array
  - **File-level errors**: Catch exceptions per file, increment skip counter, continue to next file
  - **Claim-level errors**: Logged but do not halt overall processing
- **Duplicate Handling**
  - **merge_duplicate_document_types**: True in default config (not currently checked; merge always occurs if duplicates exist)
  - **Source file tracking**: All source filenames stored in source_files array in processing_metadata
  - Multiple documents of same type automatically merged when len(documents) > 1
  - Uses latest document's timestamp and encounter_date for merged result

CLI Interface

```
bash
# Process patient folders from config paths
python 3_process-combine_text_to_json.py --limit 1

# Custom input/output directories
python 3_process-combine_text_to_json.py --input-dir "input/text_data" --output-dir "./output/"

# JSON output mode (structured events)
python 3_process-combine_text_to_json.py --json-output

# Verbose logging
python 3_process-combine_text_to_json.py --verbose
```

- **Key Flags**
  - --json-output: Enables JSON event stream on stdout, human messages on stderr
  - --limit N: Process only N patient folders
  - --verbose: Debug-level logging including file processing details
  - --dry-run: Validate configuration without processing

Processing Performance

Script 3 output enters Script 4 identically to Script 2 output, enabling unified analysis regardless of data source (synthetic vs real claims). The pure text processing approach:

- Adds minimal overhead

- Preserves original content without transformation artifacts
- Enables both LLM analysis and audit review of source text
- Supports high-volume preprocessing without API costs

7.4. Script 4: Claims Analyzer

Script 4 performs two-stage LLM extraction and aggregation to produce final actuarial variables from structured claim documents. Stage 1 processes each document independently extracting document-level features with confidence scores. Stage 2 aggregates Stage 1 features across all claim documents, applying compound scoring and conflict resolution to synthesize final assessments.

Script 4 handles both synthetic claims (from Scripts 1-2) and real claims (from Script 3) through unified JSON input format. The two-stage architecture enables document specialization (medical vs adjuster vs settlement documents require different extraction schemas) while maintaining claim-level consistency through aggregation. Compound scoring, temporal weighting, and conflict resolution produce reliable, auditable variable assessments.

The script processes claims sequentially. Each claim follows the Stage 1 → Stage 2 pipeline before proceeding to the next claim, ensuring token limits are respected while maintaining processing context.

Architecture Components

Processing Classes:

- **ClassProcessStage1**: Orchestrates document-level extraction, manages document-type-specific prompts, handles extraction failures, and obtains document-level confidence scores
- **ClassProcessStage2**: Orchestrates claim-level aggregation, implements compound scoring algorithm, executes conflict resolution protocol, generates synthesis rationales
- **ClassOrchestrator**: Main pipeline controller coordinating Stage 1 → Stage 2 flow, I/O operations, cloud uploads, batch processing

Supporting Classes

- **ClassVariableDefinitions**: Provides variable specifications, valid values, feature importance weights for scoring, cross-variable validation rules
- **ClassCalculateScore**: Implements compound scoring combining confidence, document type weight, temporal decay, feature importance
- **ClassValidateStage2**: Validates outputs against schemas, deduplicates variables, checks cost category alignment, generates quality flags
- **ClassLoadSchemas**: Manages function calling schemas for Stage 1 (document-type-specific) and Stage 2 (aggregation), injects rationale fields dynamically
- **ClassBuildPrompt**: Constructs prompts from templates, embeds variable definitions, includes compound score rankings for Stage 2
- **ClassExtractDate**: Extracts document dates from text using multiple format patterns, returns latest date found, calculates document age
- **ClassInputOutput**: Handles file loading, JSON validation, output writing with standardized naming conventions
- **ClassCloudStorage**: Optional GCS integration for uploading Stage 1 and Stage 2 outputs to cloud storage

Stage 1 Processing (Document-Level Extraction)

- **Input**: JSON from Script 2 or 3 containing claim with multiple documents
- **Process**: For each document in claim:
    1. **Determine document type** (phone_transcript, medical_provider_letter, adjuster_notes_initial, settlement_adjuster_notes, claimant_statement, clinical_note)
    2. **Load document-type-specific function calling schema** from templates/function_schemas/stage1_{document_type}.json
    3. **Inject rationale fields** into schema based on configuration (full, selective, or disabled mode)
    4. **Build extraction prompt**
        - Document type context and instructions
        - Variable definitions with extraction criteria
        - Document text content
        - Rationale requirements (verbatim vs derived)
    5. Call LLM with function calling to enforce schema compliance
    6. Parse extraction results validating against Stage 1 schema
    7. Extract document date using ClassExtractDate, calculate document age and recency factor
    8. Calculate document-level confidence scores per variable
    9. Store Stage 1 output as ClassOutputStage1 dataclass
- **Output**: Stage 1 JSON per document containing:
    - **extracted_features**: Variables with values, rationales, rationale types (verbatim/derived), confidence scores
    - **narrative_analysis**: Natural language summary of document findings
    - **confidence_score**: Overall document extraction quality
    - **processing_metadata**: Model used, timestamps, token usage
    - **document_date, document_age_days, document_weight, recency_factor**: Temporal weighting components
- **Error Handling**: Failed extractions logged with context, document flagged but processing continues, partial extractions accepted if critical variables present

Stage 2 Processing (Claim-Level Aggregation)

- **Input**: All Stage 1 ClassOutputStage1 objects for a claim
- **Process**
    1. **Load Stage 2 function calling schema** from templates/function_schemas /stage2_aggregation.json
    2. **Inject rationale fields** for aggregated variables (rationale explaining aggregation logic, stage1_sources list, conflicting_sources if conflicts detected)
    3. **Calculate compound scores for all Stage 1 features using ClassCalculateScore**:
        - Base confidence score from Stage 1 extraction
        - Document type weight (medical records weighted higher for clinical variables)
        - Temporal weight (recent documents weighted higher)
        - Feature importance weight (variable-specific document relevance)
    4. Build Stage 2 prompt with compound score rankings:
        - All Stage 1 features with compound scores
        - Variable definitions for aggregation context
        - Instructions for conflict detection and resolution
        - Cost validation rules (ultimate_cost_category must align with ultimate_cost_prediction)
    5. Call LLM with Stage 2 function calling schema
    6. Validate Stage 2 output using ClassValidateStage2:
        - Deduplicate variables appearing in multiple categories

    - Fix misplaced variables (move to correct category)
    - Validate ultimate_cost_category matches ultimate_cost_prediction range
    - Check settlement consistency (ultimate_cost should equal settlement_amount for settling claims)
    - Verify cost progression (ultimate_cost ≥ total_incurred_cost)
    - Generate quality flags for issues detected
  7. Extract cost predictions and risk assessments from aggregated variables
  8. Populate ClassOutputStage2 with complete actuarial_variables structure preserving rationale fields

- **Output**: Stage 2 JSON containing:
  - **actuarial_variables**: Three categories (reserving_variables, ratemaking_variables, claim_management_variables) with complete provenance
    - Each variable includes: value, rationale, rationale_type (aggregated/conflict_resolved), stage1_sources list, conflicting_sources (if applicable), confidence
  - **aggregated_narrative**: Natural language synthesis across all documents
  - **confidence_scores**: Overall, reserving, ratemaking, claim_management, data_completeness
  - **cost_predictions**: ultimate_cost_category, total_incurred_cost, estimated_outstanding_cost, ultimate_cost_prediction, reserve_adequacy
  - **risk_assessment**: risk_level, industry_risk_category, litigation_risk, fraud_risk
  - **quality_flags**: List of validation warnings requiring manual review
- **Conflict Resolution**: When multiple Stage 1 sources provide different values
  - Rank sources by compound score
  - Select value from highest-scoring source
  - Document all conflicting values in conflicting_sources array
  - Set rationale_type to "conflict_resolved"
  - Explain resolution logic in rationale field

Claim Processing Loop

FOR each claim JSON file in input directory:
1. **Load** claim JSON (validate structure)
2. **Initialize** Stage 1 outputs list
3. **STAGE 1**: FOR each document in claim
   - Process document through Stage 1 extraction
   - Extract document date and calculate temporal weights
   - Store ClassOutputStage1 object
   - Update metrics (document count, success/failure, tokens, cost)
4. **STAGE 2**: Aggregate all Stage 1 outputs
   - Calculate compound scores for all features
   - Build Stage 2 prompt with rankings
   - Call LLM for aggregation
   - Validate output and fix placement issues
   - Generate quality flags
5. **Save outputs**
   - Write Stage 1 outputs to stage1_features/ directory
   - Write Stage 2 output to final_analysis/ directory
   - Optional cloud upload to GCS
6. **Update cumulative metrics**

GENERATE processing summary report with stage breakdown, document type breakdown, performance metrics

Output Generation

- **Stage 1 Outputs**: Written to output/4_analysis-analyze_claims/stage1_features/ with naming pattern: {claim_id}_{patient_initials}_stage1.json
- **Stage 2 Outputs**: Written to output/4_analysis-analyze_claims/final_analysis/ with naming pattern: {claim_id}_{patient_initials}_analysis.json
- Stage 2 output follows structured schema with all 45+ actuarial variables organized into three categories. Each variable includes complete provenance chain enabling audit trail from raw documents through scoring to final assessment.

Key Features

- **Two-Stage Architecture**: Document specialization via type-specific schemas in Stage 1, claim-level synthesis in Stage 2 balances granularity with consistency
- **Compound Scoring**: Multi-component scoring (base confidence × document type weight × temporal decay × feature importance) systematically prioritizes most reliable sources
- **Temporal Weighting**: Recent documents receive higher confidence via stepped decay. Missing dates receive configurable penalty factor (default 0.7)
- **Conflict Resolution**: Systematic conflict detection across Stage 1 sources, resolution via compound score ranking, complete documentation of conflicting values and resolution rationale
- **Rationale System**: Configurable modes (full/selective/disabled) for extraction and aggregation explanations. Stage 1 rationales explain single-document extraction (verbatim vs derived). Stage 2 rationales explain multi-document aggregation and conflict resolution
- **Validation Engine**: Multi-level validation ensuring cost category alignment (critical for actuarial accuracy), settlement consistency, cost progression logic, variable placement correctness
- **Auditability**: Complete provenance from source documents through compound scoring to final variables, rationale fields explain all decisions, quality flags surface issues requiring manual review
- **Extensibility**: Configuration-driven design enables variable additions, scoring adjustments, schema modifications without code changes. Document type weights, temporal decay parameters, feature importance all configurable
- **Quality Controls**: Schema-enforced function calling, multi-level validation, confidence thresholds, automatic variable deduplication, manual review flagging for low-confidence or high-conflict cases

Performance Characteristics

- **Stage 1 Timing:** 30-45 seconds per **document** (varies by length and model)
- **Stage 2 Timing:** 45-60 seconds per **claim** (15 variables × 3-7 sources average)
- **Total Per-Claim:** 5-7 minutes for claim **with** 5-7 documents
- **Cost Per-Claim:** $0.10-0.20 using GPT-4o-mini
- **Token Usage:** 15K-25K tokens per claim total (Stage 1 + Stage 2)

Script 4 represents the core analytical engine applying sophisticated LLM-based analysis to extract actionable actuarial insights from unstructured claims documentation. The two-stage methodology, compound scoring, temporal weighting, and comprehensive provenance tracking ensure outputs meet actuarial standards for reliability, transparency, and auditability.

8. Deployment and Operations

This section provides practical guidance for deploying, operating, and maintaining the claims analysis system in production or research environments./

8.1. Cloud Integration

Cloud integration is optional. The system implements Google Cloud Storage (GCS) support in Script 4 through the ClassCloudStorage class, enabling cloud-based input/output for production deployments requiring centralized storage, team collaboration, or archival capabilities.

Google Cloud Platform Integration:

- **Cloud Storage for Input/Output:** Store documents in GCS buckets, read and write via Script 4's ClassCloudStorage class.
- **Service Account Authentication:** Use service accounts for automated processing with minimal required permissions (Storage Object Admin for specific buckets), rotate service keys periodically
- **Bucket Configuration:** Regional placement near compute resources to minimize latency and egress costs, enable versioning for audit compliance, lifecycle policies for automatic archival, implement **IAM** policies restricting access to authorized personnel
- **Regional Considerations:** Choose bucket region matching processing location, consider data residency regulations for PHI/PII, evaluate egress costs for cross-region data transfer

Setup Steps:

1. Create GCP project or use existing actuarial computing project
2. Enable Cloud Storage API
3. Create service account with Storage Object Admin role scoped to specific bucket(s)
4. Download service account JSON credentials
5. Configure script 4 YAML with project_id, bucket_name
6. Set environment variable pointing to credentials: GOOGLE_APPLICATION_CREDENTIALS=/path/to/credentials.json (or specify credentials_path in YAML to auto-set)
7. Set enabled: true in YAML cloud_storage section
8. Test with small dataset, monitor costs and optimize as needed

Configuration Example:

```
yaml
# config/4_analysis-analyze_claims.yaml
cloud_storage:
  enabled: true
  project_id: "actuarial-analytics-prod"
  bucket_name: "claims-analysis-data"
  credentials_path: "/path/to/credentials.json"
  stage1_output_path: "analysis/stage1/"
  stage2_output_path: "analysis/stage2/"
```

Local vs Cloud Decision: Local file system is sufficient for small-scale research or proof-of-concept. Cloud storage is recommended for production environments, multi-user teams, claims volumes >500, integration with cloud analytics (BigQuery, Looker), regulatory compliance requiring audit trails.

8.2. Performance Characteristics

Understanding expected performance and cost characteristics enables capacity planning, budget allocation, and processing time estimation.

Processing Times

Operation	Duration	Notes
Script 1 (FHIR processing)	2-5 min per bundle	Varies by encounter count, LLM model

Operation	Duration	Notes
Script 2 (document generation)	3-6 min per claim	Generates various document types per claim
Script 3 (text preprocessing)	<1 second per document	No LLM calls, pure text processing
Script 4 Stage 1	30-60 sec per document	Document-level extraction with function calling
Script 4 Stage 2	45-90 sec per claim	Claim-level aggregation across documents
Complete Pipeline	**4-10 min per claim**	**Scripts 3→4 with 5-7 documents typical**

Cost Estimates

Codebase pricing configuration (per 1K tokens, OpenAI GPT-4o-mini)

- **Input**: $0.00015
- **Output**: $0.0006

Scripts implement automatic cost tracking via ClassAPIClient with embedded token counting and cumulative cost calculation**.**

Scalability Considerations

- **Horizontal Scaling**: Claims process independently with no shared state, enabling parallel processing across multiple instances
- **Vertical Scaling**: Minimal impact - LLM API calls dominate processing time, not local compute
- **Batch Processing**: Scripts 1-2-3 support batch mode via --input-dir argument processing entire directories. Implemented via process_batch_files() and process_batch() methods with progress tracking
- **Cost Management**: Configure provider via YAML ("openai" or "ollama"). Ollama enables zero-cost local processing for development/testing
- **Rate Limits**: No rate limiting implemented in codebase. Production deployments must handle OpenAI limits externally or implement request spacing
- **Storage Costs**: JSON outputs compact (10-50 KB per claim)

8.3. Security Considerations

Security is critical when processing Protected Health Information (PHI) and proprietary claims data.

API Key Management

- **Environment Variables**: Codebase expects **OPENAI_API_KEY** environment variable (configurable via api_key_env setting). Never hardcode keys in scripts or config files.
- **Configuration**: YAML files reference environment variables but do not store keys directly
- **GCS Credentials**: Script 4 supports credentials_path setting that automatically sets GOOGLE_APPLICATION_CREDENTIALS environment variable
- **Key Rotation:** Change API keys quarterly or after personnel changes. Update keys in all environments simultaneously. Test key updates in development before production rollout
- **Permissions Scoping:** Limit API key **permissions** to required operations only. Create separate keys for development, staging, production. Revoke keys immediately when no longer needed

Data Privacy Protocols

- **De-identification Requirement:** Prototype assumes input data is pre-de-identified per organizational policies. Production requires HIPAA-compliant processing if handling PHI. Verify LLM provider has signed Business Associate Agreement (BAA) for PHI processing
- **LLM Provider Data Retention:** OpenAI enterprise tier: Data not used for model training, retained 30 days then deleted. **Ollama** local models: Data never leaves organizational infrastructure. Verify provider policies meet organizational compliance requirements

- **Secure Transmission**: All API calls use HTTPS (enforced by OpenAI SDK and google-cloud-storage library)
- **Audit Logging**: Scripts implement comprehensive logging via ClassLogger with file-based output. Log rotation and retention policies must be configured externally. Enable Cloud Storage access logging if using GCS
- **Data Minimization:** Process only documents necessary for actuarial analysis. Exclude unnecessary clinical detail from LLM prompts. Implement data retention policies deleting inputs/outputs after analysis complete

Cloud Security Best Practices

- **IAM Roles with Least Privilege:** Script 4 ClassCloudStorage uses service account authentication via credentials JSON file. Grant Storage Object Admin role scoped to specific buckets with minimum required permissions. Review and audit IAM policies quarterly
- **Bucket Encryption:** Configure in GCP Console, not managed by scripts. Enable encryption on all Cloud Storage buckets. Use Google-managed or customer-managed encryption keys.
- **VPC Service Controls:** Implement VPC Service Controls if available to restrict data exfiltration. Configure security perimeters around sensitive resources. Monitor for policy violations
- **Access Monitoring:** Enable Cloud Audit Logs in GCP Console. Set up alerting for unusual access patterns. Review access logs monthly for anomalies
- **Data Retention:** Configure GCS lifecycle policies in GCP Console, not managed by scripts

Compliance Considerations

Organizations processing PHI must ensure HIPAA compliance. Verify LLM providers have signed BAAs. System does not implement HIPAA controls. Compliance is organizational responsibility requiring external safeguards, risk assessments, and documented security procedures.

8.4. Testing and Validation

Comprehensive testing ensures system reliability, extraction accuracy, and production readiness.

Testing Strategy

- **Dry Run Mode**: Scripts 1 and 3 support --dry-run flag for configuration validation without LLM API calls. Returns mock data for testing infrastructure without costs.
- **Batch Processing**: All scripts implement process_batch_files() and process_batch() methods for directory-level testing with progress tracking
- **Input Validation**: Script 2 implements ClassValidateInput validating Script 1 output structure and required fields
- **Variable Validation**: Script 4 implements ClassValidateStage2 validating outputs against variable definitions, deduplicating variables, fixing misplacements
- **Limit Flag**: Scripts support --limit N ar for processing subset of encounters or files during testing.

Quality Controls

- **Quality Flags**: Script 4 generates quality flags automatically via _generate_quality_flags() method, as described in 5.4.3 Quality Flags Structure.
- **Validation Warnings**: ClassValidateStage2 generates warnings for:
  - Duplicate variables across categories
  - Misplaced variables (wrong category)
  - Definition violations (invalid values, out-of-range)
  - Consistency issues (cost progression, settlement mismatches)

Validation Procedures

- **Input Structure Validation**: Script 2 validates Script 1 output has required fields (claim_id, patient_initials, patient_metadata, encounters, processing_metadata) before processing
- **Stage 2 Output Validation**: Script 4 validates final outputs via validate_stage2_output():
  - Deduplicates variables appearing in multiple categories
  - Moves misplaced variables to correct categories
  - Validates individual variable values against definitions
  - Checks cross-field consistency (cost categories, settlement amounts)
  - Generates quality flags for issues
- **Quality Flag Reporting**: Script 4 logs quality flags with claim_id for tracking

Development/Testing

- **Use --dry-run flag** (Scripts 1, 3) to validate configuration without API costs
- Test with **--limit N** on small file batches (all scripts)
- Use **Ollama** provider (provider: "ollama" in YAML) for zero-cost local testing (all scripts)
- Process directories using batch mode via **--input-dir** (all scripts)
- Review **quality flags** in Script 4 output logs
- Examine **validation warnings** for systematic issues

Continuous Monitoring

Production deployments require external monitoring infrastructure. System generates quality flags and validation warnings.

9. Web User Interface (Prototype)

The system includes an optional web-based user interface providing browser-based access to pipeline functionality. The web UI is a development convenience enabling non-technical users to execute scripts without command-line expertise. **Current Status**: UI framework implemented with Script 1 (FHIR Bundle Processor) fully functional. Scripts 2-4 have placeholder pages but require backend integration.

9.1. Overview and Current Status

What's Implemented: Script 1 fully functional through web interface. Users can select FHIR bundle files, configure processing parameters via YAML editor, launch jobs, monitor real-time processing logs via Server-Sent Events, track token usage and costs, and access completed results. Flask backend handles Script 1 execution, log streaming, configuration editing, and result retrieval.

What's Not Implemented: Scripts 2-4 have placeholder UI pages displaying "[Placeholder for future development]" with config button only. Backend has no execution endpoints for these scripts - the script_map in /api/execute-script contains only 'process_bundle': '1_data-process_fhir_bundle.py'. Full implementation requires adding script mappings and execution handlers mirroring Script 1 pattern. Framework exists but deprioritized for core pipeline development.

Purpose: UI provides development convenience eliminating command-line barriers for non-technical team members, visual monitoring replacing log file inspection, and centralized YAML configuration editing. Not intended to replace command-line execution for production batch processing..

9.2. Architecture and Technology Stack

Technology Choices

- **Frontend:** Frontend: React 18.3.1 with component-based architecture, Vite 5.3.1 for development builds and bundling, Tailwind CSS 3.4.1 for styling, Axios 1.6.5 for HTTP requests, React Router 7.9.3 for SPA routing
- **Backend:** Flask (Python) providing REST API, Server-Sent Events (SSE) for real-time log streaming, subprocess execution for script launching with threading
- **Deployment:** Single-machine deployment (frontend served from Flask in production), development mode runs separate servers (Flask on :5000, Vite on :3000), production serves built static files from app/frontend/dist/

API Structure: The Flask backend exposes RESTful endpoints.

- POST /api/execute-script - Launch job (currently supports scriptType: 'process_bundle' only)
- GET /api/jobs - List all jobs with status
- GET /api/jobs/:jobId - Get specific job details
- DELETE /api/jobs/:jobId - Cancel/remove job
- GET /api/config/:configFile - Retrieve YAML configuration
- POST /api/config/:configFile - Update YAML configuration
- GET /api/input-files - List available input files by config
- GET /api/preview-file - Preview file contents

Script 2-4 endpoints require further implementation.

9.3. Implemented Features (Script 1)

- **File Selection Interface**: Users browse FHIR bundle JSON files from configured input_folder. Single file selection per execution. File list dynamically loaded via /api/input-files based on config. No batch processing (use CLI for multiple files).

- **Configuration Editor**: YAML configuration displayed in editable text area with save to disk. No syntax validation - invalid YAML causes backend errors.
- **Real-Time Log Streaming**: Live log output during processing. Server sends log lines as data events with newline preservation. Frontend displays in scrollable pre element with auto-scroll.
- **Job Status Monitoring**: Job tracking with status values: 'running', 'completed', 'failed'. Jobs persist in memory until server restart or manual deletion.
- **Cost Tracking Display**: Extraction from script stdout via regex patterns matching "Total cost: $X.XX", "Total tokens: X", "API calls: X", "Encounters processed: X". No detailed per-call breakdown - aggregated totals only.

9.4. Potential Development Roadmap

- **Scripts 2-4 Backend Completion:** Implement Flask endpoints mirroring Script 1 pattern. Add subprocess execution for Scripts 2-4. Integrate **log** streaming for all scripts. Enable configuration management for all components.
- **Batch Processing Enhancements:** Queue **multiple** claims for sequential processing. Parallel job execution with resource management. Batch status dashboard showing all active jobs. Automatic retry on transient failures.
- **Results Visualization:** Interactive display of extracted variables. Confidence score visualization with color coding. Source document highlighting **showing** extraction evidence. Variable population dashboards for claim batches. Comparative analysis across multiple claims.
- **Enhanced Monitoring:** Real-time performance dashboards (throughput, latency). Cost analytics with budget tracking and alerts. Quality metrics **visualization** (accuracy, confidence distributions). Error rate tracking and categorization.

The web UI enhances usability for development and demonstration but is not required for production operation. Organizations preferring command-line or programmatic execution can ignore the UI entirely.

10. Extension and Enhancement

This section describes extension points enabling system customization and outlines improvements for future development.

10.1. Extension Points

The system architecture provides clean extension points for common customization needs.

10.1.1. Adding New LLM Providers

1. Create new provider class implementing ClassLLM abstract interface

```
python
class ClassLLMClaude(ClassLLM):
    def call_llm(self, messages: List[Dict], model: str = None,
                 max_tokens: int = None, temperature: float = 0.0,
                 call_type: str = "general") -> Dict:
        # Implement using Claude API
        # Return: {'content': str, 'usage': {...}, 'cost': float,
        #          'response_time': float, 'model': str}

    def call_llm_with_function(self, messages: List[Dict], schema: Dict,
                               function_name: str, model: str = None,
                               max_tokens: int = None, temperature: float = 0.7,
                               call_type: str = "structured") -> Dict:
        # Implement structured output (function calling or equivalent)
        # Return: {'success': bool, 'function_args': dict, 'usage': {...},
        #          'cost': float, 'response_time': float, 'model': str}

    def get_cost_summary(self) -> Dict:
        # Return: {'total_cost': float, 'total_calls': int,
        #          'average_cost_per_call': float, 'total_tokens': int,
        #          'total_time': float}
```

2. Register provider in ClassCreateLLM.create_provider() factory method in llm_providers.py
3. Add provider configuration section to YAML files (api_settings section)
4. Implement provider-specific retry logic and error handling
5. Add pricing configuration for cost tracking
6. Test with sample prompts ensuring output compatibility

No changes required to business logic. Provider abstraction isolates API differences.

10.1.2. Adding New Document Types

1. Define document type in configuration (e.g., 4_analyze-claims.yaml)

```
yaml
document_types:
  incident_report:
    max_tokens: 1800
    temperature: 0.3
    priority: 2
    weight: 0.8
    specialized_template: "template_incident_report.py"
    function_schema_file: "stage1_incident_report.json"
```

2. Create document-specific extraction template in templates/specialized/template_incident_report.py specifying variables to extract and extraction guidance
3. Create JSON function schema in templates/function_schemas/stage1_incident_report.json defining the structured output format

4. Update module_variable_definitions.yaml to add the new document type to document_variable_mapping section, defining which variables are relevant for this document type with their feature importance weights
5. Add test cases validating extraction from new document type

System automatically incorporates new document types in Stage 1 processing (ClassStage1Extraction) and Stage 2 aggregation (ClassStage2Aggregation) without code changes to core business logic.

10.1.3. Adding New Actuarial Variables

1. Define variable in module_variable_definitions.yaml

```
yaml
stage1_variables:
  fraud_indicators:
    name: "Fraud Risk Assessment"
    tier: "standard"
    category: "investigative"
    definition: |
      Assessment of potential fraud indicators based on:
      - Timeline inconsistencies
      - Exaggerated symptom descriptions
      - Treatment pattern anomalies
      - Prior claim history flags
    data_type: "categorical"
    valid_values:
      - "None_Identified"
      - "Low_Risk"
      - "Moderate_Risk"
      - "High_Risk"
    validation_rules:
      - "Must be one of the specified valid values"
    examples:
      - value: "Moderate_Risk"
        context: "Minor timeline inconsistencies noted"
```

2. Add variable to document_variable_mapping section defining which document types should extract this variable and their feature importance weights

```
yaml
document_variable_mapping:
  adjuster_notes_initial:
    fraud_indicators: 0.9
  medical_provider_letter:
    fraud_indicators: 0.5
```

3. Add variable to Stage 1 extraction schemas (templates/function_schemas/stage1_*.json) for relevant document types
4. Add variable to Stage 2 aggregation schema (templates/function_schemas/stage2_aggregation.json) in appropriate category
5. Update validation rules in schemas if cross-field dependencies exist

System automatically incorporates new variables through ClassVariableDefinitions loader, which injects definitions into prompts based on document type relevance. The ClassLoadSchemas component dynamically includes new variables in function calling schemas.

10.2. Potential Improvements

Performance Optimization:

- **Batch API Calls**: Use OpenAI batch API for non-urgent processing (50% cost reduction, 24-hour latency). Queue claims for overnight processing, retrieve results next morning

- **Prompt Compression**: Reduce token usage through prompt engineering. Use more concise variable definitions without sacrificing clarity. Eliminate redundant context in prompts
- **Parallel Processing**: Implement multi-claim parallel processing respecting rate limits. Use job queue (Celery, RQ) for distributed processing. Scale horizontally with multiple workers. Current implementation processes files sequentially
- **Enhanced Caching**: Expand beyond current schema caching in ClassLoadSchemas to cache LLM responses for repeated documents (exact text matches). Cache template expansions to reduce redundant processing

Feature Extensions

- **Multi-Language Support**: Add extraction templates for non-English claims. Implement language detection in text preprocessing. Provide translated variable definitions
- **Structured Data Integration**: Direct integration with claims databases (not just file-based). Read from SQL databases or APIs for input. Write results to data warehouses for analytics. Current system operates on file-based inputs/outputs
- **Confidence Calibration and Reproducibility**: Automated confidence score calibration based on expert reviews. Measure output variance across multiple runs on identical inputs to quantify stochasticity. Adjust thresholds dynamically based on accuracy feedback. Implement deterministic mode (temperature=0) for exact reproducibility in production. Current system uses temperature=0.3, reducing but not eliminating variance
- **Temporal Cost Tracking**: Temporal Cost Tracking: Add explicit valuation dates for cost variables (incurred_cost_as_of_date, outstanding_cost_as_of_date, ultimate_cost_as_of_date). Implement date extraction logic handling format variations, conflicts, and missing values. Essential for actuarial loss development and reserve adequacy analysis. Current implementation captures document timestamps but not cost-specific valuation dates
- **Severity Progression Tracking**: Implement initial_severity and final_severity fields with dates and claim maturity milestones (reported, settlement). Track severity changes over claim lifetime with rationale stamps. Current Stage 2 uses weighted aggregation (settlement: 0.98, medical: 0.95) producing single final value. Stage 1 preserves temporal data but Stage 2 does not expose progression
- **Active Learning**: Flag most informative claims for expert review (high uncertainty). Incorporate expert feedback improving future extractions. Prioritize review of claims most impacting models

Web UI Enhancements

- **Complete Script 2-4 Integration**: Finish backend implementation for all scripts. Unified job management across pipeline
- **Advanced Visualization**: Interactive variable exploration. Confidence score heatmaps. Document provenance graphs showing source chains for Stage 2 aggregations
- **User Management**: Role-based access control (reviewers, operators, admins). Audit logging of user actions. Multi-user collaboration features
- **Batch Processing Dashboard**: Queue management for large claim volumes. Progress tracking across batches. Cost and performance analytics. Extend current batch processing capabilities with web-based monitoring

10.3. Research Directions

Model Improvements

- **Fine-Tuned Models**: Fine-tune smaller models (GPT-3.5, local models) on labeled claims data for improved accuracy at lower cost. Build organization-specific models capturing terminology and patterns. Current system uses general-purpose models (GPT-4o-mini)
- **Ensemble Methods**: Combine multiple LLM outputs (majority voting, weighted averaging). Use different models for different variable types (clinical vs investigative). Current architecture supports provider abstraction enabling this approach
- **Retrieval-Augmented Generation**: Incorporate external knowledge bases (medical literature, legal precedents). Provide LLMs with reference materials improving consistency
- **Prompt Optimization**: Systematic A/B testing of extraction prompts. Automated prompt engineering using meta-prompting techniques. Current template system (ClassLoadTemplates) provides foundation for systematic testing

Workflow Enhancements

- **Human-in-the-Loop**: Interactive extraction with real-time expert feedback. Active learning prioritizing uncertain extractions for review. Gradual automation as confidence improves
- **Explanation Generation**: Enhance current rationale system (optional per-variable explanations in shared_settings.rationale_system) to generate comprehensive natural language explanations of variable assessments. Help actuaries understand LLM reasoning
- **Anomaly Detection**: Flag claims with unusual variable combinations. Detect outliers requiring special handling. Identify potential data quality issues beyond current validation rules

Integration Capabilities

- **Actuarial Software Connectors**: Direct integration with Arius, ResQ, Ratabase. API endpoints for programmatic access. Standardized data exchange formats. Current system outputs JSON suitable for integration
- **Reporting Automation**: Generate actuarial reports from extracted variables. Template-based report generation. PDF/Excel output for distribution
- **Real-Time Processing**: Stream processing for claims as they arrive. Low-latency extraction for time-sensitive decisions. Event-driven architecture. Current implementation processes files in batch mode

These extension points and planned improvements provide a roadmap for system evolution while maintaining architectural integrity. The modular design (ClassConfig, ClassLoadSchemas, ClassLoadTemplates, ClassVariableDefinitions) ensures enhancements can be implemented incrementally without destabilizing core functionality in ClassStage1Extraction and ClassStage2Aggregation.

This completes Part II (Implementation Reference) of the TAMD. Part I covered conceptual architecture, design philosophy, and methodology. Part II provided implementation specifications for infrastructure, scripts, deployment, and extensions. The appendices provide detailed reference material for configuration, schemas, algorithms, and code navigation.

Appendix

Appendix A: Complete Script 4 Configuration Reference

This appendix provides the complete configuration reference for Script 4 (Claims Analyzer), documenting the current implementation structure with all available options, examples, and configuration patterns.

A.1 Script 4 Configuration Structure Overview

Reference: The complete, authoritative configuration is maintained in config/4_analysis-analyze_claims.yaml. This section provides an annotated structural overview; consult the actual file for current values and implementation details.

Configuration Organization

The Script 4 configuration file organizes settings into the following hierarchical structure:

1. **Logging Configuration (logging)**
   - Log levels (DEBUG, INFO, WARNING, ERROR)
   - Console and file output formatting
   - Debug options for API prompts, responses, and processing details
2. **API Settings (api_settings)**
   - Provider selection: OpenAI or Ollama
   - Model configurations for stage1 and stage2
   - Retry settings with exponential backoff
   - Provider-specific configurations:
     - **openai**: Model IDs and pricing per 1K tokens
     - **ollama**: Base URL and local model selection
3. **Processing Settings (processing)**
   - Default input/output directories
   - Stage enablement flags (stage1_enabled, stage2_enabled)
4. **Stage 1 Settings (stage1_settings)**
   - Document-level feature extraction configuration
   - Function calling enablement for structured outputs
   - Template organization (category and specialized templates)
   - Model parameters (max_tokens, temperature)
5. **Stage 2 Settings (stage2_settings)**
   - Claim-level aggregation configuration
   - Orchestrator template selection
   - Function schema for final analysis
   - Model parameters optimized for aggregation
6. **Shared Settings (shared_settings)**
   - Templates directory location
   - Rationale system configuration:
   - Mode: "full", "selective", or "disabled"
   - Selective fields list for high-value variables
7. **Document Types Configuration (document_types)**
   - Six document type definitions, each with:
     - Processing parameters (max_tokens, temperature)
     - Priority (1-3, where 1 is highest)
     - Weight (0.6-1.0, representing source authority)

    - Specialized template and function schema references
- Document types: phone_transcript, medical_provider_letter, adjuster_notes_initial, settlement_adjuster_notes, claimant_statement, clinical_note

8. **Temporal Weighting Configuration (temporal_weighting)**
    - Time-based decay for document relevance
    - Decay function selection (stepped, exponential)
    - Stepped decay thresholds (days and decay factors)
    - Missing date handling strategy
    - Document-specific overrides
9. **Compound Scoring Configuration (compound_scoring)**
    - Multi-factor source prioritization system
    - Component toggles: document_weight, confidence_score, recency_factor, feature_importance
    - Default values for missing scores
    - Threshold settings for minimum and low scores
    - Prompt inclusion settings
10. **Cloud Storage (cloud_storage)**
    - Optional Google Cloud Storage integration
    - GCS configuration (project_id, bucket_name, credentials)
    - Cloud paths for stage outputs
    - Performance settings (batch size, parallel operations)
11. **Output Structure (output_structure)**
    - Directory organization for stage1 features, stage2 analysis, reports, temp files
12. **Filename Templates (filename_templates)**
    - Naming conventions for feature extraction, final analysis, reports, and logs
13. **Performance Monitoring (performance)**
    - Tracking flags for API usage, processing time, confidence scores
    - Report generation settings
14. **Error Handling (error_handling)**
    - Retry configuration (max_retries, delay)
    - Continue-on-error behavior
    - Partial results preservation
15. **Quality Control (quality_control)**
    - Confidence flagging thresholds (category-level and variable-level)
    - Required and recommended document types
    - Conflict and missing document detection
16. **Extraction Templates (extraction_templates)**
    - Document type-specific extraction categories
    - Future customization framework
17. **Validation Rules (validation)**
    - Input validation (structure, document types, required fields)
    - Output validation (schema compliance, required fields)
    - Rationale validation (presence, types, source citations)
18. **Cost Tracking (cost_tracking)**
    - Tracking granularity (by stage, document type, claim)
    - Budget alerts (optional)
    - Cost reporting frequency

19. Experimental Features (experimental)

- Advanced conflict resolution
- Multi-model comparison
- Incremental processing
- Schema and template caching

20. Key Design Principles

- **Provider Abstraction**: Seamless switching between OpenAI and Ollama
- **Two-Stage Processing**: Document-level extraction (Stage 1) followed by claim-level aggregation (Stage 2)
- **Source Prioritization**: Multi-dimensional scoring combining document authority, confidence, recency, and feature importance
- **Template Specialization**: Document type-specific extraction templates with dedicated function schemas
- **Quality Assurance**: Comprehensive validation, confidence thresholding, and rationale tracking

A.2 Temporal Weighting Configuration Examples

The temporal weighting system applies time-based decay to document relevance. The current implementation uses a stepped decay function.

Standard Configuration

```yaml
temporal_weighting:
  enabled: true
  decay_function: "stepped"

  stepped_decay:
    thresholds:
      - days: 30
        factor: 1.0      # Current (0-30 days) - full weight
      - days: 90
        factor: 0.9      # Recent (31-90 days) - 90% weight
      - days: 180
        factor: 0.8      # Moderately old (91-180 days) - 80% weight
      - days: 365
        factor: 0.7      # Old (181-365 days) - 70% weight
      - days: 999999
        factor: 0.6      # Very old (365+ days) - 60% weight

  missing_date_strategy: "conservative"
  missing_date_factor: 0.7

  document_overrides:
    settlement_adjuster_notes:
      decay_function: "none"
      fixed_factor: 1.0
```

A.3 Compound Scoring Variations

The compound scoring system combines multiple factors to prioritize sources during aggregation.

Full Scoring

yaml

```yaml
compound_scoring:
  enabled: true
  use_document_weight: true          # Base document type authority
  use_confidence_score: true         # LLM extraction confidence
  use_recency_factor: true           # Temporal weighting
  use_feature_importance: true       # Variable-specific importance

  missing_confidence_default: 0.5
  missing_recency_default: 0.7
  missing_importance_default: 0.5

  minimum_compound_score: 0.15       # Reject sources below this threshold
  low_score_threshold: 0.4           # Flag sources below this for review

  include_scores_in_prompt: true
  score_presentation: "detailed"
```

A.4 Feature Importance Configuration

Feature importance weights are defined in the variable definitions system and map specific actuarial variables to document type/field combinations.

```yaml
feature_importance:
  injury_severity_category:              # Injury severity - clinical sources prioritized
    medical_provider_letter: 1.0         # Highest - clinical diagnosis
    clinical_note: 0.95                  # High - current clinical assessment
    adjuster_notes_initial: 0.6          # Medium - adjuster evaluation
    claimant_statement: 0.4              # Lower - subjective perception

  total_medical_costs:                   # Total medical costs - financial documents prioritized
    settlement_adjuster_notes: 1.0       # Authoritative final amount
    adjuster_notes_initial: 0.9          # Current assessment
    medical_provider_letter: 0.7         # Medical estimate
    phone_transcript: 0.3                # Initial mention

  surgery_required:                      # Surgery required - medical sources prioritized
    medical_provider_letter: 1.0         # Clinical recommendation
    clinical_note: 0.95                  # Treatment plan
    adjuster_notes_initial: 0.6          # Adjuster note
    claimant_statement: 0.4              # Patient mention

  litigation_risk:                       # Litigation risk - adjuster assessment prioritized
    adjuster_notes_initial: 1.0          # Professional assessment
    settlement_adjuster_notes: 0.95      # Settlement context
    claimant_statement: 0.6              # Attorney involvement
    phone_transcript: 0.5                # Initial indicators
```

A.5 Document Type Weights and Configuration

The current implementation defines document types with weights, priorities, and processing parameters.

Document Type Priority and Weight Guide

Priority Levels:

- **Priority 1**: Highest - processed first, most critical for analysis
- **Priority 2**: High - important for complete analysis
- **Priority 3**: Medium - valuable but not critical

Weight Interpretation:

- **1.0**: Maximum authority - definitive source
- **0.85-0.95**: High authority - strong evidence
- **0.7-0.8**: Medium authority - reliable with context
- **0.6-0.65**: Lower authority - subjective or initial information

```
yaml
document_types:
  medical_provider_letter:          # Medical provider letters - highest authority
    max_tokens: 2000
    temperature: 0.2
    priority: 1                     # Highest priority
    weight: 1.0                     # Maximum authority
    specialized_template: "template_medical_provider.py"
    function_schema_file: "stage1_medical_provider.json"

  settlement_adjuster_notes:        # Settlement adjuster notes - authoritative for final values
    max_tokens: 1800
    temperature: 0.3
    priority: 1                     # Highest priority
    weight: 0.9                     # High authority
    specialized_template: "template_settlement_notes.py"
    function_schema_file: "stage1_settlement_notes.json"

  adjuster_notes_initial:           # Initial adjuster notes - high priority for claim setup
    max_tokens: 1800
    temperature: 0.3
    priority: 2                     # High priority
    weight: 0.7                     # Medium-high authority
    specialized_template: "template_adjuster_notes.py"
    function_schema_file: "stage1_adjuster_notes.json"

  clinical_note:                    # Clinical notes - medium priority
    max_tokens: 2000
    temperature: 0.2
    priority: 2                     # High priority
    weight: 0.85                    # High authority
    specialized_template: "template_clinical_note.py"
    function_schema_file: "stage1_clinical_note.json"

  phone_transcript:                 # Phone transcripts - initial assessment
    max_tokens: 1500
    temperature: 0.3
    priority: 3                     # Lower priority
    weight: 0.7                     # Medium authority
    specialized_template: "template_phone_transcript.py"
    function_schema_file: "stage1_phone_transcript.json"
```

```
claimant_statement:                # Claimant statements - subjective source
  max_tokens: 1500
  temperature: 0.4
  priority: 3                      # Lower priority
  weight: 0.6                      # Lower authority (subjective)
  specialized_template: "template_claimant_statement.py"
  function_schema_file: "stage1_claimant_statement.json"
```

Appendix B: JSON Schemas

This appendix documents the JSON data structures for Script 4 (Claims Analyzer) inputs. See Section 5.3 for conceptual data flow overview and Section 7.4 for validation implementation.

B.1 Input Schema Structure Examples

Reference: Script 4 accepts JSON input from multiple upstream sources. The examples below demonstrate the complete structure from Script 2 (FHIR + synthetic documents). For Script 3 text file outputs, see simplified structure in Section 7.3.

Complete Example: Script 2 Output (FHIR + Synthetic Documents)

This example shows the full structure with both FHIR clinical notes and Script 2 generated synthetic documents:

```
json
{
  "claim_id": 8667856175,                  // REQUIRED: Unique identifier (string or integer)
  "patient_initials": "CO",                // Recommended: Two uppercase letters for anonymization
  "patient_metadata": {                    // Optional: Demographics (not used in analysis)
    "age": "69",
    "gender": "Female",
    "name": "Cathie710 Lizbeth716 O'Hara248"
  },
  "encounters": [                          // REQUIRED: Array of documents (minimum 1)

    // ===== FHIR Clinical Note (from Script 1) =====
    {
      "encounter_id": "5332c46a-84c3-1390-a421-af7bbd780c5a",  // Recommended: Unique ID (UUID format
for FHIR)
      "document_type": "clinical_note",    // Recommended: Enables specialized template
      "document_text": "**Clinical Note**\n\n**Patient Name:** [Patient Name]\n**Date of Encounter:**
November 12, 1984\n**Chief Complaint:** Patient presents for scheduled wellness visit with focus on
management of essential hypertension...\n\n**Assessment:** Documented history of essential hypertension,
BP 140/90 mmHg...\n\n**Plan:**\n1. Lifestyle Education: DASH diet, exercise, stress management\n2.
Monitoring: Home BP monitoring, follow-up in 3 months\n...",
      "encounter_date": "1984-11-12",      // Recommended: ISO 8601 format, enables temporal weighting
      "source_file": "synthea_native",     // FHIR metadata
      "encounter_metadata": {              // Script 1/2 only: FHIR-specific metadata
        "resource_type": "Encounter",
        "synthea_type": "wellness",
        "synthea_name": "Encounter Module Scheduled Wellness",
        "classification": {                // Script 2 relevance classification
          "is_relevant": true,
          "confidence": 0.85,
          "reason": "Essential hypertension is a chronic condition that can lead to significant long-term healthcare
costs if not managed properly.",
          "category": "moderate_cost",
          "cost_indicators": [
            "chronic condition management",
            "potential for complications",
            "medication costs",
            "monitoring and follow-up visits"
          ]
        },
        "processing_timestamp": "2025-10-25T03:16:32.536373"
      }
    },

    // ===== FHIR Clinical Note with Legacy Field Name =====
    {
      "encounter_id": "90871069-c971-92e5-98b6-c5157f354e43",
```

```
    "document_type": "clinical_note",
    "clinical_note": "**Clinical Note**\n\n**Patient Name:** [Patient Name]\n**Date of Encounter:** September 8, 1992\n**Encounter Type:** Inpatient - Tubal Ligation Surgery\n\n**Chief Complaint:** Patient presents for elective tubal ligation procedure...\n\n**Assessment:** Patient evaluated preoperatively, good overall health, no contraindications...\n\n**Plan:**\n1. Laparoscopic tubal ligation under general anesthesia\n2. Post-operative monitoring and pain management\n3. Follow-up in 1-2 weeks\n...",
    "encounter_date": "1992-09-08",
    "source_file": "synthea_native",
    "encounter_metadata": {
     "resource_type": "Encounter",
     "synthea_type": "inpatient",
     "synthea_name": "Tubal_Ligation_Surgery_Encounter",
     "classification": {
      "is_relevant": true,
      "confidence": 0.9,
      "reason": "Surgical procedure (tubal ligation) classified as high-cost treatment with potential ongoing medical implications.",
      "category": "high_cost",
      "cost_indicators": [
       "surgical procedure",
       "hospital stay",
       "anesthesia",
       "post-operative care",
       "potential complications"
      ]
     },
     "processing_timestamp": "2025-10-25T03:16:45.457762"
    }
   },

   // ===== Synthetic Document: Medical Provider Letter =====
   {
    "date": "2025-10-28T12:45:34.635591",       // Synthetic doc: generation timestamp
    "type": "other_text_data",                  // Script 2 internal classification
    "document_type": "medical_provider_letter",
    "document_text": "[Your Clinic Name]\n[Date: January 15th, 2024]\n\n[Insurance Company Name]\nAttn: Claims Adjuster\n\nRE: Claim Number: CL123456\nClaimant: C.O.\n\nDear Claims Adjuster,\n\nI am writing to provide a comprehensive medical summary for my patient, C.O., a 69-year-old retired teacher who sustained injuries from a slip and fall incident on November 12, 1984...\n\n**Clinical Summary:** Fractured right hip with surgical intervention on November 15, 1984 by Dr. Emily White...\n\n**Treatment Plan:** Physical therapy from Nov 1984 to Feb 1985, achieved MMI on Dec 15, 1986...\n\n**Prognosis:** Significant functional recovery, returned to part-time work March 1986...\n\nSincerely,\nDr. Emily White, MD",
    "generation_metadata": {                    // Script 2 only: Synthetic generation metadata
     "generated_timestamp": "2025-10-28T12:45:34.635591",
     "generator": "synthetic_document_generator",
     "document_display_name": "Medical Provider Letter",
     "generation_method": "function_calling",
     "model_name": "gpt-4o-mini-2024-07-18",
     "provider": "openai",
     "response_time": 17.96204289997695,
     "tokens": {
      "prompt_tokens": 842,
      "completion_tokens": 736,
      "total_tokens": 1578
     }
    }
   },

   // ===== Synthetic Document: Phone Transcript =====
   {
    "date": "2025-10-28T12:45:54.762947",
    "type": "other_text_data",
    "document_type": "phone_transcript",
```

```
    "document_text": "[Phone ringing]\n\n**Insurance Rep:** Hello, this is Sarah from Acme Insurance. Am I speaking with C.O.?\n**Claimant:** Yes, this is C.O.\n**Insurance Rep:** I'm following up on claim CL123456. Could you tell me what happened?\n**Claimant:** Sure, it was November 12th, 1984. I slipped on wet pavement at City Park and fell hard. Fractured my right hip...\n**Insurance Rep:** Who was your treating physician?\n**Claimant:** Dr. Emily White did the hip surgery on November 15th. Then Dr. Robert Green for physical therapy...\n...",
    "generation_metadata": {
     "generated_timestamp": "2025-10-28T12:45:54.762947",
     "generator": "synthetic_document_generator",
     "document_display_name": "Initial Phone Call Transcript",
     "generation_method": "function_calling",
     "model_name": "gpt-4o-mini-2024-07-18",
     "provider": "openai",
     "response_time": 20.116078499995638,
     "tokens": {
      "prompt_tokens": 867,
      "completion_tokens": 932,
      "total_tokens": 1799
     }
    }
   },

   // ===== Synthetic Document: Settlement Adjuster Notes =====
   {
    "date": "2025-10-28T12:46:13.719438",
    "type": "other_text_data",
    "document_type": "settlement_adjuster_notes",
    "document_text": "**Settlement Adjuster Notes for Claim C.O.**\nClaim Number: CL123456\nDate of Evaluation: 01/15/2024\n\n**Medical Treatment Summary:** Fractured right hip with sprained wrist from slip and fall on Nov 12, 1984. Surgery by Dr. Emily White on Nov 15, 1984. PT completed Feb 15, 1985. MMI achieved Dec 15, 1986.\n\n**Total Expenses:**\n- Surgical costs: $20,000\n- Physical therapy (15 sessions): $1,500\n- Total medical: $21,500\n\n**Wage Loss:** Est. $9,230 (12 weeks @ $40k annual salary)\n\n**Settlement Analysis:** Total damages $30,730. Recommend opening negotiations at $35,000...",
    "generation_metadata": {
     "generated_timestamp": "2025-10-28T12:46:13.719438",
     "generator": "synthetic_document_generator",
     "document_display_name": "Settlement Adjuster Notes",
     "generation_method": "function_calling",
     "model_name": "gpt-4o-mini-2024-07-18",
     "provider": "openai",
     "response_time": 18.938607299991418,
     "tokens": {
      "prompt_tokens": 883,
      "completion_tokens": 671,
      "total_tokens": 1554
     }
    }
   },

   // ===== Synthetic Document: Claimant Statement =====
   {
    "date": "2025-10-28T12:46:28.651267",
    "type": "other_text_data",
    "document_type": "claimant_statement",
    "document_text": "I, C.O., would like to describe the incident that happened on November 12th, 1984, at City Park. It was rainy and I slipped on wet pavement. I fell hard on the ground and couldn't get up. The pain was unbearable...\n\nI was diagnosed with a fractured right hip. Dr. Emily White performed surgery on Nov 15, 1984. Recovery was long - several months of PT with Dr. Robert Green. Couldn't do simple tasks, missed social gatherings. Returned to part-time work in 1986, declared stable Dec 15, 1986...\n\nThis incident changed my life forever. Even now at 69, I still have occasional pain flare-ups...",
    "generation_metadata": {
     "generated_timestamp": "2025-10-28T12:46:28.651267",
     "generator": "synthetic_document_generator",
     "document_display_name": "Claimant Statement",
```

```
      "generation_method": "function_calling",
      "model_name": "gpt-4o-mini-2024-07-18",
      "provider": "openai",
      "response_time": 14.917966500041075,
      "tokens": {
       "prompt_tokens": 868,
       "completion_tokens": 565,
       "total_tokens": 1433
      }
     }
    },

    // ===== Synthetic Document: Initial Adjuster Notes =====
    {
     "date": "2025-10-28T12:46:43.560759",
     "type": "other_text_data",
     "document_type": "adjuster_notes_initial",
     "document_text": "INITIAL ADJUSTER NOTES\nClaim Number: CL123456\nClaimant: C.O.\nDate of
Review: 1/15/24\n\n**Incident Details:**\n- Mechanism: slip and fall on wet pavement, City Park, Springfield,
11/12/84\n- Witnesses: John Smith & Mary Johnson\n\n**Medical Info:**\n- Primary injury: fractured right
hip, surgery by Dr. Emily White 11/15/84\n- Secondary: sprained wrist\n- PT by Dr. Robert Green, completed
02/15/85\n- MMI achieved 12/15/86, stable, no further surgery\n\n**Liability:** Park management
responsibility for wet pavement conditions\n\n**Next Steps:**\n- Follow up with medical providers for
complete records\n- Schedule IME for 2/15/24\n- Prepare for settlement discussions pending IME results...",
     "generation_metadata": {
      "generated_timestamp": "2025-10-28T12:46:43.560759",
      "generator": "synthetic_document_generator",
      "document_display_name": "Initial Adjuster Notes",
      "generation_method": "function_calling",
      "model_name": "gpt-4o-mini-2024-07-18",
      "provider": "openai",
      "response_time": 14.898194900015369,
      "tokens": {
       "prompt_tokens": 866,
       "completion_tokens": 605,
       "total_tokens": 1471
      }
     }
    }
   ],

   "processing_metadata": {                    // Recommended: Audit trail and provenance
    "source_type": "fhir_processor",           // Indicates Script 1/2 pipeline
    "bundle_file": "Cathie710_Lizbeth716_O'Hara248_c5fd333e-21c3-d902-1d5c-d43d2fe253ae.json",
    "processing_timestamp": "2025-10-25T03:16:45.458761",    // ISO 8601 format
    "total_encounters_evaluated": 3,
    "relevant_encounters": 2,
    "processing_stats": {                      // Upstream processing statistics
     "total_evaluated": 3,
     "relevant_encounters": 2,
     "classification_errors": 0,
     "note_generation_errors": 0,
     "api_calls": 0,
     "total_cost": 0.0,
     "skipped_irrelevant": 1
    },
    "cost_summary": {                          // LLM API cost tracking
     "total_cost": 0.00519795,
     "total_calls": 25,
     "total_tokens": 13044,
     "average_cost_per_call": 0.000207918
    },
    "processor_version": "enhanced_v1.0",
    "synthetic_documents_added": {             // Script 2 augmentation summary
```

```
      "timestamp": "2025-10-28T12:46:43.560759",
      "generator": "synthetic_document_generator",
      "documents_added": 5,                    // 5 synthetic docs added to 2 FHIR encounters
      "total_encounters": 7                    // Final count: 2 FHIR + 5 synthetic = 7 total
    }
  }
}
```

Field Reference Summary

Field Path	Type	Required	Purpose	Script 4 Behavior
claim_id	string|int	YES	Unique claim identifier	Used in logging and output filenames
patient_initials	string	No	Two uppercase letters (^[A-Z]{2}$)	Used in output file naming; claim_id used if missing
patient_metadata	object	No	Demographics (age, gender, name)	Optional context; not used in analysis
encounters	array	YES	Documents array (min 1)	Processed sequentially by Stage 1
encounters[].document_text	string	YES	Document content (min 50 chars)	Primary content field
encounters[].clinical_note	string	YES	Legacy FHIR field name	Aliased to document_text
encounters[].document_type	string	No	Document classification (see B.1.2)	Selects specialized template; "unknown" if missing
encounters[].encounter_id	string	No	Unique encounter identifier	Auto-generated if missing (content hash)
encounters[].encounter_date	string	No	ISO 8601 date (YYYY-MM-DD)	Enables temporal weighting
encounters[].date	string	No	ISO 8601 datetime (synthetic docs)	Alternative date field for Script 2
encounters[].encounter_metadata	object	No	FHIR-specific metadata	Script 1/2 only; includes classification
encounters[].generation_metadata	object	No	Synthetic doc metadata	Script 2 only; includes token usage
processing_metadata	object	No	Upstream provenance	Recommended for audit trail
processing_metadata.source_type	string	No	"fhir_processor" or "text_file_processor"	Indicates pipeline origin

B.2 Document Type Source Matrix

The following table shows which pipeline scripts produce each document type.

Document Type	Script 1 (FHIR)	Script 2 (Synthetic)	Script 3 (Text)	Script 4 Support
clinical_note	Yes	No	Manual	Yes
phone_transcript	No	Yes	Yes	Yes
medical_provider_letter	No	Yes	Yes	Yes
adjuster_notes_initial	No	Yes	Yes	Yes
settlement_adjuster_notes	No	Yes	Yes	Yes
claimant_statement	No	Yes	Yes	Yes

Source Details:

- **Script 1 (FHIR Processor)**: Extracts clinical_note from FHIR Encounter/DocumentReference resources
- **Script 2 (Document Augmentation)**: Generates **5 synthetic document types** (all except clinical_note)
- **Script 3 (Text Preprocessor)**: Can process any document type from text files with manual classification
- **Script 4 (Claims Analyzer)**: Accepts all **6 types** with specialized extraction templates

Configuration Reference:

- **Document type weights and priorities**: config/4_analysis-analyze_claims.yaml → document_types
- **Specialized templates**: templates/specialized/template_*.py
- **Function schemas**: templates/function_schemas/stage*.json

Important Notes:

- **Unknown document_type** values are processed with generic templates (reduced extraction quality)
- Each type has different processing parameters (max_tokens, temperature) configured in Script 4
- **Document weights** range from 0.6 (claimant_statement) to 1.0 (medical_provider_letter)
- **Priority levels**: 1 (highest), 2 (high), 3 (medium) affect processing order

B.3 Validation Rules and Behavior

Script 4 Validation Approach

Script 4 implements a lenient validation strategy focused on pipeline flexibility. Only critical fields are strictly validated; most validation failures generate warnings but allow processing to continue.

Strict Validation (Input Rejection)

The following conditions cause immediate processing failure.

- **Required Top-Level Fields**
  - claim_id must be present (string or integer)
  - encounters array must be present and non-empty (minimum 1 encounter)
- **Document Content**
  - Each encounter must have document text in at least one field: document_text, clinical_note, notes, or text
  - Encounters with no content in any of these fields are skipped with warning
- **Data Quality**
  - LLM's structured output must be ≥50 characters for meaningful analysis
  - Input must be valid JSON structure

Lenient Validation (Warnings Only)

The following issues generate warnings but allow processing to continue:

- **Unknown Document Types**
  - Unrecognized document_type values → processed with generic template
  - Warning logged: "Unknown document type, using generic template"
- **Missing Optional Fields**
  - Missing patient_initials → output files use claim_id only
  - Missing document_type → defaults to "unknown"
  - Missing encounter_date → recency_factor is explicitly set to a default value
  - Missing processing_metadata → informational message only
- **Invalid Enum Values**
  - Invalid gender values → logged but ignored
  - Invalid source_type → logged but processing continues

Automatic Normalization

Script 4 automatically normalizes inputs for maximum compatibility.

- **Document Text Field Aliasing**

```
python
# Script 4 checks fields in this order (4_analysis-analyze_claims.py)
document_text = (
  encounter.get('document_text') or            # Preferred field name
  encounter.get('clinical_note') or    # FHIR legacy format
  encounter.get('notes') or            # Script 2 synthetic docs
  encounter.get('text') or             # Generic fallback
  ''                                   # Last resort (triggers error)
)
```

- **Generated Identifiers**
  - Missing encounter_id → auto-generated as a sequential string combining the claim_id and the document index
  - Prevents duplicate processing of identical documents

- **Default Values**
  - Missing document_type → defaults to "unknown" (uses generic template)
- **Date Handling**
  - Future encounter_date → document age set to 0 days (not rejected, no warning)
  - Uses max(0, age_days) to prevent negative ages
  - Affects temporal weighting calculation (future dates get maximum recency weight)

Critical Output Validation

Script 4 validates LLM-generated outputs with strict criteria that cause processing failure.

- **Stage 1 Output Validation**
  - **LLM output length**: Structured JSON output must be ≥50 characters
  - **Failure behavior**: Logs ERROR and rejects entire document processing
  - **Note**: This validates LLM output quality, not input document length
- **Stage 2 Cost Category Validation**
  - **Cost category mismatch**: If ultimate_cost_prediction and ultimate_cost_category are inconsistent
  - **Failure behavior**: Logs ERROR and adds COST_CATEGORY_MISMATCH quality flag (CRITICAL)
  - **Example**: ultimate_cost_prediction: 50000 but ultimate_cost_category: "Low" triggers error

Business Logic Validation

Script 4 performs consistency checks but does not reject on failure:

- **Consistency Checks**
  - total_encounters_evaluated ≥ relevant_encounters (if both present)
  - processing_stats.total_evaluated should match encounters array length
  - tokens.total_tokens = prompt_tokens + completion_tokens (if all present)
  - Inconsistencies logged as warnings but do not prevent processing
- **Temporal Weighting (Automatic Calculation)**
  - Document age from encounter_date automatically affects recency_factor calculation
  - Example: Documents >90 days old may have recency reduced from 1.0 to 0.9
  - This is automatic calculation, not a validation warning

Validation Behavior Summary

Validation Type	Behavior	Example
Critical	Reject input	Missing claim_id or encounters
Quality	Warn, continue	Document text <50 characters
Enum	Warn, use default	Unknown document_type → "unknown"
Consistency	Log info only	Mismatched encounter counts
Auto-fix	Normalize silently	clinical_note → document_text

Validation Configuration

Validation behavior is controlled by config/4_analysis-analyze_claims.yaml.

```yaml
validation:
  # Input validation
  validate_input_structure: true        # Enable structure validation
  validate_document_types: true         # Warn on unknown types
  require_claim_id: true                # Reject if missing
  require_encounters: true              # Reject if missing

  # Output validation
  validate_stage1_schema: true
  validate_stage2_schema: true
```

```
validate_required_fields: true

# Rationale validation (when enabled)
validate_rationale_presence: true
validate_rationale_types: true
require_source_citations: true
```

See Also

- Section 7.4 for Script 4 implementation details
- Section 5.5 for validation framework architecture

Document Version History

Version	Date	Author	Changes
1.0	October 28, 2025	Technical Team	Initial architecture documentation
1.1	October 30, 2025	Technical Team	Added Sections 2-3: Architecture Design Philosophy and Core Processing Methodology
2.0	November 24, 2025	Technical Team	Added Sections 4-7: Document Processing Pipeline, Data Structures & Schema Design, Core Infrastructure Components, Script Implementation Details
3.0	December 16, 2025	Technical Team	Added Section 8-Appendix: Deployment & Operations, Web UI, Extension Framework, Simplified Appendices A-B