UW Companion

AI-Powered Underwriting Assistant

Intelligent Document Analysis for Commercial Insurance Underwriters

Version 1.0
February 15, 2026

RAG-Powered • Hallucination Detection • Action Extraction

AIG — Commercial Insurance Technology

Table of Contents

1. Overview

UW Companion is an AI-powered document analysis platform designed specifically for commercial insurance underwriters. It enables underwriters to upload policy documents (PDF and DOCX), ask natural-language questions about document contents, and receive accurate, source-cited answers with built-in hallucination detection and automated underwriting action extraction.

The system employs a Retrieval-Augmented Generation (RAG) architecture, combining vector similarity search over LanceDB with Google Gemini large language models to deliver grounded, trustworthy responses. Every AI-generated answer is accompanied by a multi-factor hallucination score that quantifies how well the response is supported by the source documents.

Key Capabilities

- **Document Ingestion** — Parse PDF and DOCX files, extract text with page provenance
- **Smart Chunking** — Section-aware document splitting with configurable overlap
- **Vector Search** — Semantic similarity search using Gemini embeddings and LanceDB
- **RAG Chat** — Contextual question-answering grounded in uploaded documents
- **Hallucination Detection** — 4-factor scoring system for response trustworthiness
- **Action Extraction** — Automated identification of underwriting actions with priorities
- **Source Citations** — Every claim linked back to specific documents and page numbers

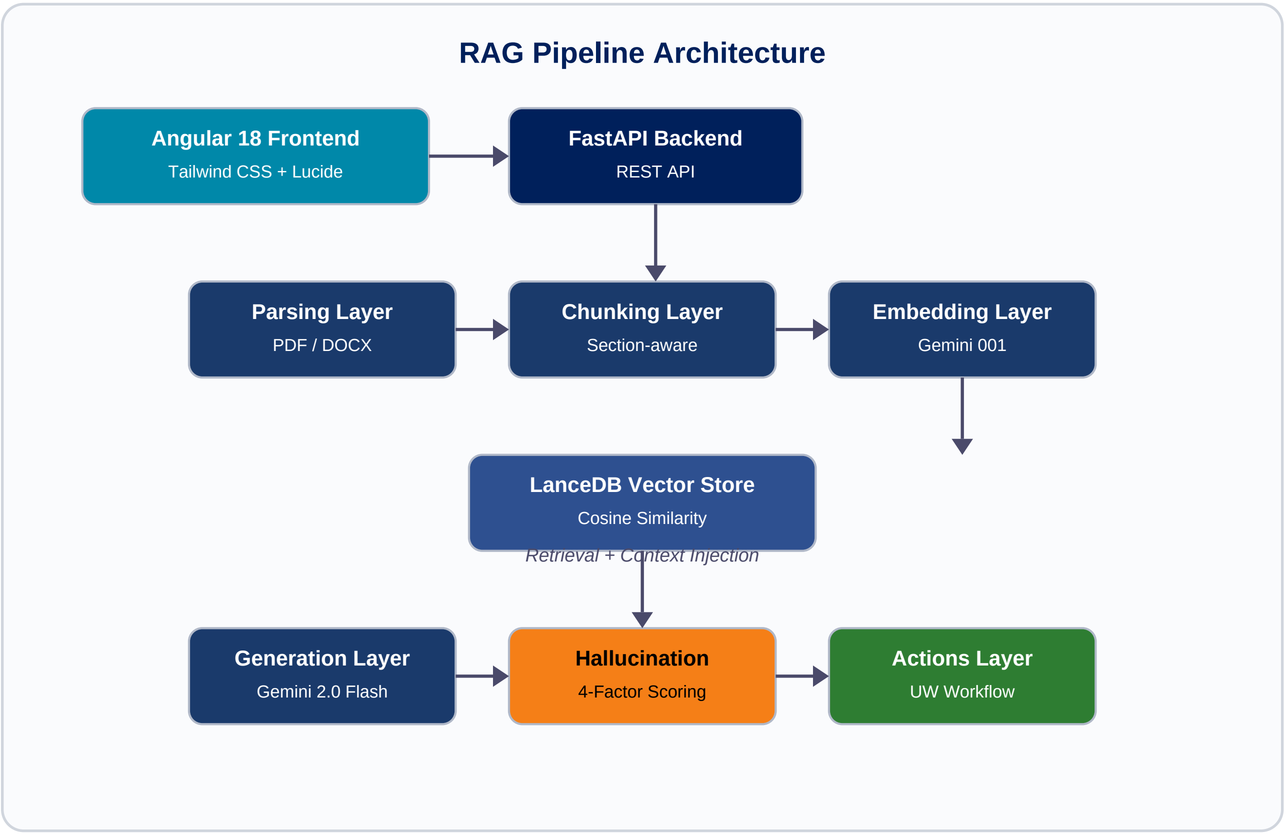
2. Architecture Overview

UW Companion follows a clean separation between the frontend presentation layer and a Python backend that implements the RAG pipeline as a series of composable layers.

Technology Stack

Component	Technology	Details
Frontend	Angular 18	Standalone components, Tailwind CSS v4, Lucide icons
Backend	Python FastAPI	Async REST API with layered architecture
Vector Database	LanceDB	In-memory / file-based vector store, cosine similarity
Chat LLM	Google Gemini 2.0 Flash	Fast, high-quality generative model
Embeddings	Gemini Embedding 001	3072-dimensional vectors, separate task types
PDF Parsing	PyPDF2	Page-by-page text extraction
DOCX Parsing	python-docx	Paragraph and table extraction with synthetic pages
Validation	Pydantic v2	Request/response schema validation

RAG Pipeline Flow



The RAG pipeline processes documents through six sequential layers: Parsing -> Chunking -> Embedding -> Vectorization (storage). At query time, the pipeline performs embedding -> vector search -> generation -> hallucination analysis -> action extraction. Each layer is independently testable and replaceable.

3. Layer Architecture

The backend is organized into seven discrete layers, each with a single responsibility. Layers communicate through well-defined interfaces (Python dicts and Pydantic models), enabling independent development, testing, and replacement.

3.1 Parsing Layer

Location: `layers/parsing/parser.py` | **Team:** Document Ingestion Team

The Parsing Layer is the entry point for document ingestion. It accepts file paths for PDF and DOCX documents and extracts structured text with page-level provenance. Each page is returned as a (page_number, text) tuple, preserving the association between content and its location in the original document.

Capabilities:

- **PDF Parsing** (PyPDF2) — Iterates through pages, extracts text per page, filters empty pages, returns list of (page_num, text) tuples
- **DOCX Parsing** (python-docx) — Extracts paragraphs and table cell content, synthesizes page boundaries (40 paragraphs per page) since DOCX lacks native pages
- **Auto-detection** — The `parse_document()` function detects format by file extension and dispatches to the correct parser

Interface: `parse_document(filepath: str) → list[tuple[int, str]]`

3.2 Chunking Layer

Location: `layers/chunking/chunker.py` | **Team:** NLP / Document Processing Team

The Chunking Layer splits parsed document text into semantically meaningful pieces sized for embedding and retrieval. It employs a section-aware strategy that preserves the logical structure of underwriting documents.

Chunking Strategy (in order of precedence):

- **Section header detection** — Recognizes patterns like SECTION, PART, ARTICLE, SCHEDULE, ENDORSEMENT, Roman numerals, and numbered headings
- **Paragraph-level splitting** — Falls back to paragraph boundaries for non-sectioned text
- **Recursive sentence splitting** — For oversized chunks, recursively splits at sentence boundaries to maintain coherence
- **Overlap injection** — Appends trailing tokens from the previous chunk to maintain context continuity across chunk boundaries

Parameter	Default	Description
CHUNK_SIZE	512 tokens	Maximum number of tokens per chunk
CHUNK_OVERLAP	64 tokens	Number of trailing tokens carried into the next chunk

Output: `list[Chunk]` where each Chunk contains chunk_id, text, source filename, page number, section header, and token estimate.

3.3 Embedding Layer

Location: `layers/embedding/gemini_embedder.py` | **Team:** ML / Embeddings Team

The Embedding Layer converts text into high-dimensional vector representations using Google's Gemini Embedding 001 model. It supports batch processing for efficient document indexing and uses differentiated task types for optimal retrieval quality.

Property	Value
Model	models/gemini-embedding-001
Vector Dimensions	3072
Batch Size	100 texts per API call
Document Task Type	retrieval_document (for indexing)
Query Task Type	retrieval_query (for search queries)

Using separate task types (`retrieval_document` vs `retrieval_query`) is critical for optimal retrieval performance. The embedding model optimizes vectors differently based on whether the text will be stored (document) or used for searching (query).

3.4 Vectorization Layer

Location: `layers/vectorization/lance_store.py` | **Team:** Data Infrastructure Team

The Vectorization Layer manages the LanceDB vector store. It handles chunk storage with embeddings, similarity search at query time, and document lifecycle management (add, list, delete). LanceDB operates as a file-based store in `/tmp/uw_companion_lancedb`.

Core Operations:

- **store_chunks()** — Embeds text chunks, creates vector records with metadata (chunk_id, text, source, page, section, document_id), upserts into LanceDB
- **search()** — Embeds the query using retrieval_query task type, performs cosine similarity search, returns top-K results with similarity scores
- **get_all_documents()** — Returns metadata for all indexed documents
- **delete_document()** — Removes all chunks belonging to a document by document_id

Parameter	Default	Description
LANCE_DB_PATH	/tmp/uw_companion_lancedb	Path to LanceDB storage directory
LANCE_TABLE_NAME	document_chunks	Name of the LanceDB table
TOP_K_RESULTS	8	Number of chunks returned per search query

3.5 Generation Layer

Location: `layers/generation/rag_generator.py` | **Team:** AI / LLM Team

The Generation Layer constructs prompts from retrieved document chunks and chat history, then generates responses using Google Gemini 2.0 Flash. It also provides action extraction prompts for the Actions Layer.

Functions:

- **generate_rag_response()** — Builds a system prompt with underwriting expert persona, injects document context with source citations, includes chat history for multi-turn conversations, enforces rules (cite sources, no approximation, flag risks)
- **extract_actions_prompt()** — Generates a structured JSON extraction prompt requesting UW actions with categories (coverage_gap, risk_flag, endorsement, compliance, pricing) and priority levels

The system prompt enforces strict grounding: the model must answer ONLY from provided context, cite specific sources and pages, use exact numbers, and explicitly flag when information is not available in the documents.

3.6 Hallucination Detection Layer

Location: `layers/hallucination/detector.py` | **Team:** AI Safety / Trust Team

The Hallucination Detection Layer is the trust and safety backbone of UW Companion. It evaluates every AI response using a 4-factor scoring system to quantify how well the response is grounded in the source documents. The output is a detailed report including per-sentence grounding analysis and flagged claims.

See Section 4 for the full algorithm specification.

3.7 Actions Layer

Location: `layers/actions/extractor.py` | **Team:** Underwriting Workflow Team

The Actions Layer extracts structured underwriting actions from AI analysis results. It uses Gemini to identify actionable items from the conversation context and parses them into validated UWAction objects.

Action Schema:

Field	Type	Values / Description
action	string	Short description of what the underwriter should do
category	enum	coverage_gap, risk_flag, endorsement, compliance, pricing
priority	enum	critical, high, medium, low
details	string	1-2 sentence explanation of the action
source_reference	string	Document and page the action relates to

The extraction uses Gemini to generate a JSON array of actions, which is then parsed and validated. Invalid categories or priorities are automatically corrected to safe defaults (risk_flag / medium). Markdown code fences in the LLM output are stripped before JSON parsing.

4. Hallucination Detection Algorithm

Every AI-generated response is scored on four complementary factors. The final hallucination score (0–100) is a weighted combination of these factors, where higher scores indicate greater trustworthiness.

Factor 1: Retrieval Confidence

Weight: 0.25 (25%)

Measures the quality of the retrieved document chunks used to generate the response. Computed as a position-weighted average of chunk similarity scores — earlier (more relevant) chunks receive higher weight. The weighting formula uses `weight[i] = 1 / (i + 1)` for the i-th chunk. The result is clamped to [0, 100].

Factor 2: Response Grounding

Weight: 0.35 (35%) — Heaviest factor

The most important factor. Evaluates each sentence in the AI response individually by computing its embedding similarity against all source chunks. For each sentence, the best matching source is identified. A sentence is considered **grounded** if its similarity exceeds the threshold of **0.65**. The per-sentence score is normalized as `min(1.0, similarity / 0.8) * 100`. The factor score is the average across all sentences.

Ungrounded sentences (below threshold) are collected into the `flagged_claims` list for underwriter review.

Factor 3: Numerical Fidelity

Weight: 0.20 (20%)

Checks whether numerical values in the AI response (dollar amounts, percentages, plain numbers, formatted numbers) can be found in the source documents. Uses regex patterns to extract numbers in formats such as `$1,000,000`, `5.5%`, `$2.5 million`. If no numbers appear in the response, the score defaults to 100 (no numerical claims to verify).

Factor 4: Entity Consistency

Weight: 0.20 (20%)

Verifies that named entities in the response (policy numbers, dates, proper names, multi-word capitalized terms) also appear in the source documents. Entity extraction uses regex patterns for: policy/form numbers (e.g., CGL-2024001), dates in multiple formats (MM/DD/YYYY, Month DD, YYYY), and capitalized proper nouns. If no entities are found in the response, the score defaults to 100.

Rating Thresholds

Score Range	Rating	Indicator	Meaning
80 – 100	LOW risk	Green	Response is well-grounded in source documents
50 – 79	MEDIUM risk	Amber	Some claims may not be fully supported
0 – 49	HIGH risk	Red	Significant hallucination detected — review carefully

```
Overall Score = (Retrieval Confidence x 0.25) + (Response Grounding x 0.35) + (Numerical
Fidelity x 0.20) + (Entity Consistency x 0.20). Result is clamped to [0, 100] and rounded to 1
decimal place.
```

5. API Reference

The UW Companion backend exposes a RESTful API via FastAPI. Base URL: `http://localhost:8000`

POST /api/documents/upload

Upload a PDF or DOCX document for processing. The document is parsed, chunked, embedded, and stored in the vector database.

Property	Details
Content-Type	multipart/form-data
Form Field	file (UploadFile) — the PDF or DOCX file
Accepted Types	.pdf, .docx, .doc
Success Response	200 OK — DocumentUploadResponse
Error Responses	400 (unsupported type, no filename, no text) | 500 (processing error)

Response Schema (DocumentUploadResponse):

Field	Type	Description
document_id	string (UUID)	Unique identifier for the uploaded document
filename	string	Original filename
num_chunks	integer	Number of chunks stored in vector DB
num_pages	integer	Number of pages extracted
status	string	"indexed" on success

GET /api/documents

List all uploaded and indexed documents.

Response: Array of DocumentInfo:

Field	Type	Description
document_id	string	Document UUID
filename	string	Original filename
num_chunks	integer	Number of indexed chunks

Field	Type	Description
num_pages	integer	Number of pages
upload_time	string (ISO 8601)	Timestamp of upload

DELETE /api/documents/{document_id}

Remove a document and all its chunks from the vector store. Also deletes the uploaded file from the server.

Property	Details
Path Parameter	document_id (string) — UUID of the document to delete
Success Response	200 OK — {"status": "deleted", "document_id": "..."}
Error Response	404 Not Found — document does not exist

POST /api/chat

Send a natural-language query and receive a RAG-generated answer with hallucination analysis and underwriting action extraction.

Request Body (ChatRequest):

Field	Type	Default	Description
query	string	(required)	The underwriting question to ask
session_id	string	"default"	Session ID for multi-turn conversation history

Response Body (ChatResponse):

Field	Type	Description
answer	string	AI-generated response grounded in document context
sources	SourceReference[]	Top 5 source chunks with text, file, page, similarity
hallucination	HallucinationReport	4-factor hallucination analysis (see Section 4)
actions	UWAction[]	Extracted underwriting actions with priorities
session_id	string	Session identifier for this conversation

HallucinationReport Schema:

Field	Type	Description
overall_score	float (0-100)	Weighted composite hallucination score
retrieval_confidence	float	Factor 1 score
response_grounding	float	Factor 2 score
numerical_fidelity	float	Factor 3 score
entity_consistency	float	Factor 4 score
sentence_details	SentenceGrounding[]	Per-sentence grounding analysis
flagged_claims	string[]	Sentences with grounding below threshold
rating	string	"low", "medium", or "high" risk

DELETE /api/chat/session/{session_id}

Clear the chat history for a specific session. Useful for starting a fresh conversation.

Property	Details
Path Parameter	session_id (string) — session to clear
Success Response	200 OK — {"status": "cleared", "session_id": "..."}

GET /health

Health check endpoint for monitoring and deployment readiness probes.

Property	Details
Success Response	200 OK — {"status": "ok", "gemini_configured": true/false}
Authentication	None required

6. Frontend Features

The UW Companion frontend is built with Angular 18 using standalone components, Tailwind CSS v4 for styling, and Lucide icons for the icon system. It provides a modern, responsive interface for underwriters.

Dashboard

- Real-time metrics display showing document count, chunk count, and system status
- Hallucination monitor with aggregate statistics across sessions
- Recent activity feed for uploaded documents and chat interactions
- Analytics view with visual representations of underwriting insights

Document Management

- Drag-and-drop document upload supporting PDF and DOCX formats
- Processing indicator showing parsing, chunking, and embedding progress
- Document list with metadata (filename, pages, chunks, upload time)
- One-click document deletion with confirmation
- Document panel component for detailed document inspection

AI Chat Interface

- Natural-language query input with command bar interface
- Streaming-style response display with markdown rendering
- Per-message hallucination gauge showing trust level (green/amber/red)
- Source references panel showing matched document chunks with similarity scores
- Flagged claims highlighting for sentences with low grounding scores
- Multi-turn conversation support with session management
- Session clearing for fresh conversations

Underwriting Actions Panel

- Automatic extraction of actionable items from AI analysis
- Priority badges with color coding (critical=red, high=orange, medium=yellow, low=green)
- Category chips for quick filtering (coverage gap, risk flag, endorsement, compliance, pricing)
- Action cards with detail expansion and source reference links
- Insight cards component for summarized analytical views

Navigation & Layout

- Sidebar navigation with icon-based nav items
- Theme service supporting light/dark mode toggle
- Responsive layout adapting to different screen sizes
- Consistent design language with AIG branding

7. Configuration Reference

All configuration is centralized in `config.py` at the backend root. Environment variables can override defaults.

Variable	Default	Description
GEMINI_API_KEY	(env var)	Google AI Studio API key for Gemini access
GEMINI_CHAT_MODEL	gemini-2.0-flash	Model used for RAG response generation and action extraction
GEMINI_EMBED_MODEL	models/gemini-embedding-001	Model used for text embedding
EMBEDDING_DIM	3072	Dimensionality of embedding vectors
LANCE_DB_PATH	/tmp/uw_companion_lancedb	File path for LanceDB storage
LANCE_TABLE_NAME	document_chunks	LanceDB table name for chunk vectors
CHUNK_SIZE	512	Maximum tokens per document chunk
CHUNK_OVERLAP	64	Overlap tokens between adjacent chunks
TOP_K_RESULTS	8	Number of chunks retrieved per search query

Hallucination Weights

The `HALLUCINATION_WEIGHTS` dictionary controls the relative importance of each hallucination detection factor:

Key	Weight	Factor
retrieval_confidence	0.25	Quality of retrieved chunks
response_grounding	0.35	Per-sentence similarity to sources (heaviest)
numerical_fidelity	0.20	Number matching between response and sources
entity_consistency	0.20	Named entity matching

8. Setup & Running

Prerequisites

- Python 3.9 or higher
- Node.js 18+ and npm (for frontend)
- A Google AI Studio API key with Gemini access

Backend Setup

1. Install Python dependencies:

```
pip install -r requirements.txt
```

2. Set your Gemini API key:

```
export GEMINI_API_KEY=your_api_key_here
```

3. Start the FastAPI server:

```
python3 -m uvicorn main:app --host 0.0.0.0 --port 8000 --reload
```

The API will be available at `http://localhost:8000`. Interactive API docs (Swagger UI) are at `http://localhost:8000/docs`.

Frontend Setup

1. Install Node.js dependencies:

```
npm install
```

2. Start the Angular development server:

```
ng serve
```

The frontend will be served at `http://localhost:4200` and will proxy API requests to the backend.

Key Dependencies (requirements.txt)

Package	Version	Purpose
fastapi	0.115.6	Async web framework
uvicorn[standard]	0.34.0	ASGI server
lancedb	0.20.0	Vector database
google-generativeai	0.8.4	Gemini API client

Package	Version	Purpose
PyPDF2	3.0.1	PDF text extraction
python-docx	1.1.2	DOCX text extraction
numpy	>=1.24.0,<2.1	Numerical computing for embeddings
pydantic	2.10.4	Data validation and schemas
pytest	8.3.4	Testing framework
httpx	0.28.1	Async HTTP client for testing

9. Feature List

Document Ingestion & Processing

- PDF document upload and text extraction (PyPDF2)
- DOCX document upload with paragraph and table extraction (python-docx)
- Page-level text provenance tracking for accurate source citations
- Automatic file format detection and parser dispatch
- Uploaded file persistence in temporary storage for lifecycle management

Smart Document Chunking

- Section-aware chunking that respects document structure
- Detection of common insurance document headers (SECTION, ENDORSEMENT, SCHEDULE, etc.)
- Recursive splitting: sections → paragraphs → sentences → token windows
- Configurable chunk size (default 512 tokens) and overlap (default 64 tokens)
- Section header prepended to each chunk for context
- Unique UUID assigned to every chunk for tracking

Vector Search & Retrieval

- 3072-dimensional Gemini embeddings for high-quality semantic search
- Differentiated embedding task types (retrieval_document vs. retrieval_query)
- Batch embedding with 100-text batches for efficient indexing
- LanceDB vector store with cosine similarity search
- Configurable top-K retrieval (default 8 chunks)
- Document-level management (add, list, delete) in the vector store

RAG-Powered AI Chat

- Natural-language question-answering over uploaded documents
- Expert underwriting system prompt with strict grounding rules
- Context injection with source document and page citations
- Multi-turn conversation support with session-based chat history
- Chat history capped at 20 messages per session for performance
- Graceful handling of empty document stores

Hallucination Detection

- 4-factor composite scoring system (retrieval confidence, response grounding, numerical fidelity, entity consistency)
- Per-sentence grounding analysis with embedding similarity comparison
- Automatic flagging of ungrounded claims (similarity below 0.65 threshold)
- Numerical extraction and cross-referencing (dollar amounts, percentages, numbers)

- Named entity extraction and source verification (policy numbers, dates, proper nouns)
- Three-tier risk rating: LOW (green), MEDIUM (amber), HIGH (red)
- Detailed HallucinationReport with sentence-level breakdown

Underwriting Action Extraction

- LLM-based extraction of structured underwriting actions
- Five action categories: coverage_gap, risk_flag, endorsement, compliance, pricing
- Four priority levels: critical, high, medium, low
- Automatic validation and safe-default correction for invalid categories/priorities
- Source reference linking to specific documents and pages
- JSON output parsing with markdown fence stripping

API & Integration

- RESTful API built with FastAPI for high performance
- Automatic OpenAPI/Swagger documentation at /docs
- CORS configured for local Angular development (ports 4200)
- Pydantic v2 request/response validation
- Health check endpoint for monitoring and deployment probes
- Session management for multi-user chat isolation

Frontend Experience

- Angular 18 with standalone components architecture
- Tailwind CSS v4 for responsive, utility-first styling
- Lucide icon system for consistent visual language
- Dashboard with real-time metrics and hallucination monitoring
- Document management panel with upload and deletion
- AI Chat with per-message hallucination gauge
- Underwriting Actions panel with priority badges and category chips
- Source reference cards with similarity scores
- Insight cards for analytical summaries
- Light/dark theme support via theme service
- Command bar interface for quick navigation

10. Testing

UW Companion uses pytest as its testing framework with httpx for async API testing.

Test Infrastructure

- **Framework:** pytest 8.3.4
- **HTTP Client:** httpx 0.28.1 (for async FastAPI test client)
- **Test Location:** `backend/tests/`

Running Tests

```
pytest
```

With verbose output:

```
pytest -v
```

With coverage:

```
pytest --cov=layers --cov=services --cov-report=term-missing
```

Recommended Test Coverage

- **Unit Tests:** Each layer should have isolated unit tests with mocked dependencies
- **Parsing Layer:** Test PDF/DOCX extraction with sample documents, empty files, corrupt files
- **Chunking Layer:** Test section detection, recursive splitting, overlap injection, configurable parameters
- **Embedding Layer:** Test batch processing, query embedding, API error handling
- **Vectorization Layer:** Test CRUD operations, search result ordering, similarity score computation
- **Generation Layer:** Test prompt construction, history injection, system prompt enforcement
- **Hallucination Layer:** Test each factor independently, composite scoring, edge cases (no numbers, no entities, empty response)
- **Actions Layer:** Test JSON parsing, category/priority validation, malformed LLM output handling
- **API Integration Tests:** End-to-end tests for each endpoint using FastAPI TestClient

End of Documentation

Generated on February 15, 2026 — UW Companion v1.0