Politecnico di Torino

Data Science and Engineering

A.a. 2024/2025

Graduation Session Dicembre 2025

# AI-Powered Claims Assessment

**Leveraging Intelligent Agents for Automated Claims Management**

Supervisors:
Prof. Paolo Garza
Eng. Edoardo Morucci
Eng. Edoardo Lardizzone

Candidate:
Matteo Bracco

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Background and Motivation

Traditional insurance claims processing depends on manual review by human adjusters who examine submitted documents, interpret policy terms, determine applicable coverage, and calculate reimbursement amounts. This manual approach results in significant operational inefficiencies, including multi-day processing times that delay customer payouts, inconsistent interpretations of ambiguous clauses that lead to disputes, and substantial labor costs associated with maintaining large teams of skilled adjusters.

Advances in Large Language Models, Retrieval-Augmented Generation, and Optical Character Recognition now create the possibility of automating large portions of this process while maintaining the accuracy, transparency, and accountability expected in regulated environments. LLMs enable natural language understanding and reasoning over complex policy language, RAG provides factual grounding through retrieval of relevant contract clauses, and OCR converts diverse document formats into machine-readable text. Yet deploying these technologies in insurance presents distinct challenges. Privacy regulations such as the GDPR require strict separation and controlled retention of personal data, regulatory compliance mandates that every automated decision include verifiable references to policy text, financial computations demand exact decimal precision to avoid monetary discrepancies, and human oversight must remain integral for ambiguous or high-impact cases. Addressing these domain-specific constraints is essential to achieving trustworthy and compliant AI-driven claims automation.

## 1.2 Problem Statement

This thesis presents the design and implementation of an AI-powered insurance claims assessment system that achieves a balance between accuracy, efficiency, privacy compliance, and regulatory explainability. The system is designed to extract information from heterogeneous documents, interpret complex policy contracts, compute precise reimbursement amounts, and generate explanations grounded in verifiable policy text, all while maintaining interactive response times and economically sustainable operating costs.

The research addresses five core technical and regulatory challenges. Document understanding focuses on hybrid OCR pipelines capable of processing diverse document types, including digital PDFs, scanned forms, and smartphone-captured images, each requiring tailored extraction methods for optimal accuracy. Semantic retrieval develops mechanisms for mapping natural language questions to relevant policy clauses, achieving a balance between precision to avoid irrelevant context and recall to ensure complete coverage. Grounded generation enforces factual integrity by constraining language model outputs to retrieved contract text and producing explicit citations, satisfying regulatory requirements for explainable and auditable AI decisions. Validation and calculation integrate deterministic financial logic with AI-driven document understanding, ensuring that deductible applications, coverage limits, and co-pay computations are transparent and numerically exact. Privacy and compliance are achieved through a dual data architecture that separates persistent policy corpora from temporary customer uploads, automatically deletes session data after processing, and maintains human oversight for ambiguous or high-stakes cases.

## 1.3 Research Objectives

The primary objective of this thesis is to design, implement, and evaluate an AI-powered insurance claims assessment system that demonstrates technical feasibility and production readiness in regulated environments. To achieve this goal, the research pursues several specific objectives. It develops a modular architecture that integrates Large Language Models, Retrieval-Augmented Generation, Optical Character Recognition, and deterministic validation logic within a privacy-compliant dual vectorstore framework that separates permanent policy data from temporary customer uploads. It implements hybrid document processing pipelines that apply type-aware extraction strategies, using native text extraction for digital PDFs and enhanced OCR for scanned or photographed documents. It adapts RAG techniques for the insurance domain by introducing explicit citation mechanisms to ensure factual grounding and conducting a systematic cost-quality tradeoff analysis comparing different model tiers. It designs deterministic validation workflows that compute reimbursements, verify coverage, and generate alerts while seamlessly integrating with AI-driven document understanding. It establishes evaluation protocols assessing OCR accuracy, retrieval faithfulness, system performance, and alignment between AI outputs and human expert judgments. Finally, it develops methodologies for generating synthetic insurance datasets that preserve structural and semantic realism while ensuring complete privacy compliance, enabling robust testing and public demonstration without exposure of personal data.

## 1.4 Contributions

The key contributions of this thesis are severalfold. It presents a production-oriented system architecture that integrates Large Language Models, Retrieval-Augmented Generation, and Optical Character Recognition within a modular design that satisfies privacy and compliance requirements through dual vectorstore separation, a pattern broadly applicable to other regulated domains beyond insurance. It demonstrates the effectiveness of hybrid document processing strategies that combine native extraction for digital documents with enhanced OCR for scanned or photographed materials, achieving superior accuracy and efficiency compared to uniform processing pipelines. It implements RAG with explicit citation mechanisms and hallucination detection, ensuring grounded and verifiable outputs suitable for use in regulated decision-making contexts. It introduces a comprehensive evaluation framework with metrics tailored to insurance applications, emphasizing faithfulness, citation accuracy, and human-AI agreement rather than generic language benchmarks. It develops a systematic methodology for generating realistic synthetic insurance data, enabling privacy-compliant experimentation and validation. Finally, it provides an empirical analysis of cost-quality tradeoffs between GPT-4 and GPT-3.5-turbo models, offering practical guidance for tiered deployment strategies that balance economic efficiency with decision quality.

## 1.5 Thesis Organization

Chapter 2 reviews the foundational technologies and prior research relevant to this work, including Large Language Models, Retrieval-Augmented Generation architectures, Optical Character Recognition methods, and vector database systems used for semantic retrieval. Chapter 3 details the system architecture and implementation, covering the backend services, RAG subsystem, OCR processing pipeline, validation and calculation logic, and the React-based frontend interface. Chapter 4 presents the empirical evaluation, measuring OCR accuracy, retrieval faithfulness, overall system performance, and alignment between AI-generated and human expert decisions. Chapter 5 concludes the thesis by summarizing key contributions, identifying current limitations, and outlining directions for future research and development.

# Chapter 2

# Related Work Technologies

## 2.1 Introduction

Claims management represents one of the most critical and complex processes within the insurance industry. It encompasses all stages following the submission of a claim, from its initial registration in the company's system to the eventual issuance of reimbursement to the client. The process is characterized by several persistent challenges, including labor-intensive document review, inconsistent assessments across cases, and lengthy processing times that hinder operational efficiency and customer satisfaction.

Recent advances in artificial intelligence, particularly in large language models (LLMs), Retrieval-Augmented Generation (RAG), and optical character recognition (OCR), provide new opportunities to address these challenges and to enhance the efficiency, consistency, and accuracy of claims processing. The integration of these technologies enables systems that can augment human decision-making or, in some cases, support fully automated processing pipelines.

This chapter analyzes the technologies and methodologies that make AI-powered claims processing possible. It begins with a review of prior work in insurance automation (chapter 2.2), followed by an examination of foundational AI technologies, including large language models (chapter 2.3), RAG architectures (chapter 2.4), and OCR-based document processing (chapter 2.5). Subsequent sections explore vector embeddings and semantic search (chapter 2.6), the LangChain framework for orchestrating LLM applications (chapter 2.7), and the design of human-in-the-loop systems (chapter 2.8). The final sections (chapter 2.9–2.10) discuss the criteria guiding technology selection and conclude the overall analysis.

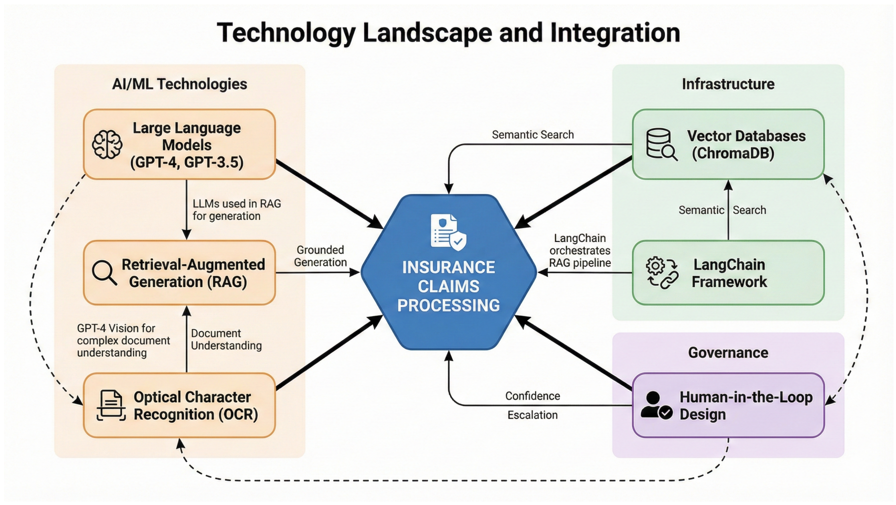
Technology Landscape and Integration
AI/ML Technologies
Large Language Models (GPT-4, GPT-3.5)
LLMs used in RAG for generation
Retrieval-Augmented Generation (RAG)
GPT-4 Vision for complex document understanding
Document Understanding
Optical Character Recognition (OCR)
Grounded Generation
INSURANCE CLAIMS PROCESSING
Semantic Search
Infrastructure
Vector Databases (ChromaDB)
Semantic Search
LangChain orchestrates RAG pipeline
LangChain Framework
Governance
Confidence Escalation
Human-in-the-Loop Design

**Figure 2.1:** Conceptual map illustrating relationships between LLMs, RAG, OCR, Vector DBs, LangChain, and Human-in-the-Loop technologies within the insurance claims processing domain

## 2.2 AI in Insurance Claims Management: State of the Art

### 2.2.1 Traditional Claims Processing Challenges

Insurance claims processing comprises a sequence of interdependent steps, each contributing to the overall management of a claim. The process begins with Document Collection, during which documents of various formats and from multiple sources are aggregated into the company's system. This is followed by Information Extraction, where relevant data are identified and extracted through parsing or other analytical methods. The next stage, Policy Verification, involves checking which policies are associated with each client to confirm eligibility and coverage. Once verified, the process advances to Damage Assessment, where the existence and extent of damage to the insured person or property are evaluated. During the Fraud Detection phase, client information is compared and cross-checked with claim details to identify potential inconsistencies or discrepancies. Finally, the Reimbursement Calculation stage determines the compensation amount owed to the claimant based on verified coverage and policy terms.

Within this context, manual workflows present several limitations that significantly affect both operational efficiency and decision consistency. High processing latency is a common issue, as the time required to complete a single claim can range from several days to multiple weeks, particularly in complex cases. Another challenge arises from inter-adjuster variability, the inconsistencies in how different claims adjusters interpret similar policy provisions, which can lead to unequal outcomes across comparable claims. Substantial labor costs further compound inefficiency, as the large number of hours required for manual review necessitates extensive workforce investment. Data entry errors, often originating from unnoticed human mistakes early in the process, can propagate throughout the workflow, producing significant downstream costs. Additionally, limited scalability poses a major operational constraint: during claim surges caused by large-scale events such as natural disasters, manual systems struggle to parallelize the additional workload effectively, leading to bottlenecks and delays.

These inefficiencies clearly reveal the limitations of traditional, fully manual implementations, which are neither fault-tolerant nor cost-effective. Consequently, beginning in the 1980s, insurance companies started to investigate and develop methods for automating claims processing, marking the early stages of research into computer-assisted decision systems within the insurance sector.

### 2.2.2 Early Automation Attempts

Early automation efforts in the insurance sector relied on systems that encoded policy logic as explicit rule sets and decision trees, commonly referred to as rule-based expert systems. While these systems were effective in handling narrowly defined scenarios, they proved brittle in practice. Any modification to policy terms or regulatory requirements necessitated extensive manual updates to the underlying rules, and cases falling outside the predefined rule coverage often resulted in processing failures. Moreover, as noted by Hayes-Roth et al. (1983) [1], maintenance costs frequently exceeded the initial development investments, making such systems economically unsustainable over time.

In the late 1990s, the first generation of optical character recognition (OCR) systems emerged [2]. These models were based on template matching and hand-crafted feature extractors, achieving acceptable performance on clean, high-quality printed documents. However, their accuracy deteriorated sharply when confronted with handwritten text, low-quality scans, or heterogeneous document layouts, all common characteristics of insurance claim materials.

The early 2000s marked a shift toward statistical machine learning approaches, including Support Vector Machines, Random Forests, and other related algorithms. These models were applied to tasks such as fraud detection and claim classification [3] [4]. While they represented a significant step forward in adaptability compared to rule-based systems, their performance remained heavily dependent on extensive feature engineering and large labeled datasets. As a result, their generalization across diverse insurance claim scenarios was limited, restricting their practical scalability in production environments.

### 2.2.3 Modern AI Technologies

In recent years, document understanding has benefited greatly from advances in deep learning, particularly through the introduction of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). These architectures enabled end-to-end document processing without the need for manual feature engineering, representing a major breakthrough in automation and scalability [5] [6]. The robustness of optical character recognition (OCR) systems improved substantially as well, particularly with the integration of Long Short-Term Memory (LSTM) layers into the Tesseract 3.0 model [7] [8], which enhanced sequential character recognition and reduced error rates. Nevertheless, challenges persist for degraded scans, handwritten documents, and heterogeneous layouts, which continue to test model generalization. The subsequent development of the Transformer architecture [9] marked another paradigm shift in natural language processing (NLP), including its applications in insurance claim management. The Transformer replaced recurrent structures with self-attention mechanisms, allowing models to process entire sequences in parallel and enabling large-scale training on massive text corpora. This innovation laid the foundation for a new generation of pre-trained models such as BERT [10], which demonstrated that transfer learning could effectively leverage general-purpose language representations for domain-specific tasks, even with limited labeled data.

Building on this foundation, the emergence of Large Language Models (LLMs) such as GPT-3 [11] and GPT-4 [12] brought unprecedented capabilities through their vast parameter counts, ranging from tens to hundreds of billions, and their ability to exhibit emergent in-context learning behaviors [13]. These models can perform novel tasks directly from example prompts without any parameter updates, a property particularly relevant in the insurance domain. By providing policy text, claim details, and customer data as contextual input, such models can generate accurate, context-aware responses without the need for costly task-specific fine-tuning.

Most recently, multimodal models such as GPT-4 Vision [12] have extended these capabilities by processing both textual and visual information within a unified framework. This development enables true multimodal AI, capable of jointly understanding text, tables, diagrams, and images. For insurance claims, where documents frequently combine textual descriptions, structured forms, and photographic evidence of damage, this integration holds significant promise. It may substantially reduce reliance on separate OCR, layout analysis, and image processing pipelines, paving the way for more efficient and holistic claim assessment systems.

### 2.2.4 Industry Applications and Research

Several commercial and research institutions have actively explored the application of artificial intelligence to claims automation. Among the most prominent examples is Lemonade Insurance, which has deployed an AI-powered chatbot designed to accelerate claims processing. The system automatically analyzes customer submissions, cross-references them with policy data, and performs anomaly detection to identify inconsistencies or potential fraud. Despite its high degree of automation, complex or high-value cases continue to require human oversight to ensure accuracy and regulatory compliance.

Similarly, Tractable has developed computer vision systems specifically tailored for vehicle damage assessment. These systems analyze photographs of damaged vehicles to estimate repair costs and have reportedly processed millions of claims annually for major insurers. By automating visual assessment, Tractable's approach significantly reduces evaluation time and supports consistent cost estimation across large claim volumes.

In academic research, recent work has begun to extend these capabilities into language-based reasoning and contract analysis. Studies by Bommarito and Katz (2022) [14] and Katz et al. (2023) [15] demonstrate that GPT-4 can extract contractual terms, identify ambiguities, and answer legal questions with performance approaching that of domain experts. Parallel research on Retrieval-Augmented Generation (RAG) has shown that grounding language model outputs in retrieved policy text can markedly reduce hallucinations compared to purely parametric generation [16] [17]. These findings highlight the growing potential of large language models and retrieval-enhanced systems to support or even partially automate policy interpretation and claim assessment.

However, despite substantial progress, existing AI-driven solutions remain limited in scope. Many commercial and research efforts target narrow claim categories, such as vehicle collisions, without achieving broader generalization across heterogeneous insurance products. Integration with legacy enterprise systems also presents persistent challenges, often constraining the deployment of end-to-end AI workflows. Furthermore, regulatory requirements for explainability and auditability are not consistently addressed, and human-in-the-loop processes are frequently implemented as ad hoc additions rather than as integral components of system design. As a result, the full potential of AI for comprehensive, transparent, and reliable claims automation remains only partially realized.

## 2.3 Large Language Models (LLMs)

### 2.3.1 the Transformer Architecture

Large Language Models (LLMs) form the computational core of modern AI-powered chatbots and represent one of the most transformative developments in contemporary data science. To understand their operation, it is essential to analyze the architecture on which they are built. LLMs are founded on the Transformer architecture introduced by Vaswani et al. (2017) in Attention Is All You Need. Unlike earlier sequence models based on Recurrent Neural Networks (RNNs) [18] or Long Short-Term Memory (LSTM) networks [19], transformers rely exclusively on self-attention mechanisms. This innovation allows parallel processing of input sequences and enables efficient scaling to massive datasets, fundamentally changing how neural networks handle sequential data by shifting from inherently sequential computations to fully parallelizable operations.

The self-attention mechanism computes a weighted representation for each token by considering its relationship with every other token in the sequence. Formally, given query (Q), key (K), and value (V) matrices, each obtained as learned linear projections of input embeddings, the attention operation is defined as:

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.1}$$

where d_k denotes the dimensionality of the key vectors. This formulation allows the model to capture long-range dependencies efficiently, overcoming the bottlenecks inherent in recurrent architectures. However, the quadratic computational complexity $O(n^2)$ with respect to sequence length n poses practical challenges for very long documents. To address this, several extensions have been proposed, including sparse attention mechanisms [20] and linear attention approximations [21], which significantly reduce memory and time requirements while maintaining representational fidelity.

Transformers employ multi-head attention, wherein multiple attention heads operate in parallel, each learning distinct relational patterns such as syntactic dependencies, semantic associations, or coreference relationships. For instance, GPT-3 utilizes 96 attention heads distributed across 96 layers, enabling the model to form hierarchically specialized representations at varying levels of abstraction [11]. The outputs from these parallel heads are concatenated and processed through a position-wise feed-forward network, which expands the hidden dimensionality, typically by a factor of four, before projecting back to the original size. Activation functions such as the Gaussian Error Linear Unit (GELU) [22] introduce nonlinearity and further enrich representational capacity.

Because the attention mechanism itself does not encode positional information, transformers incorporate positional encodings to preserve word order and syntactic structure. The original architecture used sinusoidal functions of varying frequencies, allowing the model to infer relative and absolute positions. Modern LLMs, however, often employ learned positional embeddings or relative position encodings [23], which improve flexibility and generalization. These positional signals enable the model to distinguish between semantically distinct sentences such as "the customer filed a claim" and "a claim filed the customer," ensuring that ordering information is retained throughout processing.

Finally, architectural components such as residual connections [**he2015deepresiduallearning** and layer normalization [24] are essential for stabilizing the training of deep transformer networks. Residual pathways facilitate gradient flow across layers, while normalization prevents internal covariate shifts, both of which are crucial for training models with dozens or even hundreds of layers. More recent improvements, such as Pre-Layer Normalization [24], have further enhanced training stability, allowing models to scale beyond one hundred billion parameters while maintaining convergence and robustness.

## 2.3.2 Pre-training and Fine-tuning Paradigm

Modern Large Language Models (LLMs) are typically developed through a two-stage training process consisting of pre-training and fine-tuning [25] [26]. During the pre-training phase, the model is exposed to massive text corpora comprising hundreds of billions to trillions of tokens collected from heterogeneous sources such as web pages, books, scientific publications, and code repositories. For instance, GPT-3 was trained on approximately 300 billion tokens drawn from datasets including Common Crawl, WebText2, Books1, Books2, and Wikipedia [11]. Achieving this scale necessitates distributed training across thousands of GPUs over several weeks or months, representing computational investments in the order of millions of dollars.

Pre-training relies on self-supervised learning objectives, which enable models to learn from raw text without manual labeling. The most common objective is causal language modeling, where the model predicts each subsequent token given all preceding tokens in the sequence. Formally, for a token sequence $[t_1, t_2, \ldots, t_n]$, the model maximizes the conditional probability

$$P(t_i \mid t_1, \ldots, t_{i-1}) \tag{2.2}$$

for each position i. This objective compels the model to internalize linguistic regularities, semantic relationships, and general world knowledge implicitly present in the corpus. Before training, text is tokenized using subword segmentation algorithms such as Byte Pair Encoding (BPE) [27] or SentencePiece [28], which balance vocabulary compactness and representational efficiency. GPT models typically employ vocabularies of 50,000 to 100,000 tokens, where each token corresponds to common words, frequent subwords, or rare character fragments.

The fine-tuning stage adapts the pre-trained model to specific domains or downstream tasks using smaller, labeled datasets. During this phase, model parameters are selectively adjusted to optimize performance on domain-relevant objectives while retaining the general linguistic and reasoning capabilities acquired during pre-training. In the insurance context, fine-tuning on policy documents, claims data, and regulatory text can enhance understanding of specialized terminology and improve task-specific performance. However, fine-tuning introduces potential risks such as catastrophic forgetting, where the model's general knowledge deteriorates as it specializes [29]. Mitigating this requires careful hyperparameter control to balance domain adaptation and generalization.

As model scales have surpassed 100 billion parameters, a new capability, emergent in-context learning, has appeared. This property allows models to perform previously unseen tasks directly from examples embedded within the prompt, without any modification to model parameters [13]. The emergence of such few-shot learning behaviors has been shown to scale predictably with model size, computational resources, and dataset diversity [30].

For insurance applications, these capabilities suggest that effective automation may not require full fine-tuning. Instead, providing relevant policy text and customer data as contextual input can enable accurate question answering, information extraction, and reasoning directly within the prompt. This retrieval-augmented approach offers substantial operational advantages: policy or regulatory updates can be reflected instantly through changes to the knowledge base rather than through model retraining, and general-purpose LLMs can serve multiple insurance lines without the need for distinct fine-tuned variants.

### 2.3.3 GPT Family: Evolution and Capabilities

The Generative Pre-trained Transformer (GPT) series developed by OpenAI exemplifies the state of the art in autoregressive language modeling, showcasing progressive improvements in scale, reasoning ability, and alignment with human intent. The evolution from GPT-3 to GPT-4 Vision highlights how architectural refinements and training innovations have driven increasingly sophisticated language understanding, reasoning, and multimodal capabilities.

GPT-3, introduced in 2020, contained approximately 175 billion parameters and was trained on about 300 billion tokens drawn from large and diverse corpora [11]. Its architecture comprised 96 layers with 96 attention heads per layer and a hidden dimensionality of 12,288. GPT-3 demonstrated strong few-shot learning performance across a wide range of natural language tasks, including translation, arithmetic, question answering, and text completion, often matching or surpassing specialized models without task-specific fine-tuning. The model's performance scaled with the number of in-context examples: zero-shot (task description only), one-shot (a single example), and few-shot (multiple examples) configurations produced progressively higher accuracy. Nonetheless, GPT-3 remained text-only and was prone to hallucinations, occasionally generating plausible yet factually incorrect outputs, especially for knowledge-intensive queries or multi-step reasoning tasks.

GPT-3.5-turbo represented a major refinement optimized through Reinforcement Learning from Human Feedback (RLHF) to enhance helpfulness, truthfulness, and safety [31]. The RLHF process consists of three stages: supervised fine-tuning on human-authored demonstrations, training a reward model to predict human preference rankings, and finally, optimizing the base model with Proximal Policy Optimization (PPO) [32] to align its outputs with these preferences. This alignment process substantially reduced harmful or irrelevant responses and improved instruction following, making GPT-3.5-turbo more suitable for real-world production systems. The model achieved a favorable balance between quality, latency, and cost, processing requests faster and more efficiently than GPT-3 while maintaining high performance in structured extraction, classification, and summarization tasks central to insurance workflows.

GPT-4, released in 2023, introduced notable gains in reasoning, factual reliability, and adherence to user intent (OpenAI, 2023). Although its internal architecture remains proprietary, GPT-4 demonstrated superior performance on complex reasoning benchmarks, including the Uniform Bar Exam (90th percentile), the LSAT (88th percentile), and multiple medical and legal domain evaluations. For insurance applications, these improvements translate into more accurate interpretation of policy language, better handling of conditional clauses, and enhanced multi-step reasoning for tasks such as computing coverage limits and deductibles. Some GPT-4 variants also feature extended context windows of up to 32,000 tokens, depending on the API tier and configuration. This larger context capacity enables comprehensive processing of long policy documents, entire claims histories, or multiple related files within a single prompt, reducing the need for truncation or summarization.

GPT-4 Vision (GPT-4V) extends these capabilities by incorporating multimodal understanding, processing both text and images within a unified framework (OpenAI, 2023). The model employs a visual encoder that converts images into token-like representations compatible with the text transformer, allowing joint reasoning over visual and textual information. In the context of insurance claims processing, this multimodal capability is particularly valuable: GPT-4V can analyze damage photographs to assess severity, extract structured information from complex forms containing mixed text and checkboxes, interpret tables with spanning cells and nested headers, and reason over relationships between textual descriptions and accompanying visual evidence. Unlike traditional OCR pipelines that separately handle text extraction and layout interpretation, GPT-4V integrates both modalities, enabling more accurate and semantically consistent document understanding.

### 2.3.4 LLM Capabilities Relevant to Insurance

Several capabilities of Large Language Models are particularly pertinent to claims automation, each addressing specific challenges within the insurance workflow. Natural language understanding enables these models to interpret complex policy language that includes legal terminology, conditional clauses, and exceptions. Policies frequently contain nested conditions such as "coverage applies if the incident occurs within the policy period and the insured was not engaged in commercial use of the vehicle, unless explicitly covered under the commercial rider endorsement," and LLMs parse such structures by attending to logical connectives like if, unless, and except while maintaining coreference over long passages.

They also disambiguate polysemous terms according to context, distinguishing, for instance, between premium as a payment amount and premium as a quality tier, or between deductible in medical versus property insurance usage. This contextual competence extends naturally to domain-specific jargon such as exclusions, riders, endorsements, and subrogation without requiring explicit definitions.

Information extraction then converts unstructured text into structured data suitable for downstream processing. Given a claim description such as "On January 15th, I was driving on Via Roma when another vehicle ran a red light and hit my front bumper, causing approximately 2,000 euros in damage," an LLM can produce well-formed fields, incident_date = "2024-01-15," location = "Via Roma," damage_type = "collision," estimated_cost = 2000, fault = "other_party." Whereas traditional rule-based approaches would require brittle, field-specific patterns and still falter on phrasing variants like "the 15th of January" or "two thousand euros," LLMs generalize across expression patterns through learned semantic representations. The same capability applies to adjuster notes, medical reports, and police statements, enabling automated data entry that formerly demanded manual review.

Question answering over policy documents represents a core application that relies on the model's ability to locate relevant clauses, evaluate applicability conditions, and reason over exceptions with the policy text as context. When asked, "Does policy X cover flood damage for customer Y?", the model must identify whether customer Y holds policy X, verify coverage types, inspect natural-disaster clauses, and consider exclusions such as Acts of God. Unlike keyword search, which might retrieve any section mentioning flood, semantic understanding recognizes that "water damage from natural disasters" can imply flood coverage, or that an exclusion could preclude it, thereby performing multi-hop reasoning that mirrors how human adjusters analyze claims.

Text generation then supports coherent explanations, summaries, and customer communications. After determining eligibility, the system can produce a clear report such as: “Your claim for windshield replacement is approved. Coverage applies under comprehensive insurance with a 150 euros deductible. Your reimbursement is 350 euros (500 euros repair cost minus 150 euros deductible). Processing will complete within 5 business days.” This requires integrating calculation outputs with policy terms and standard communication practices. The same mechanism adapts tone and register for different audiences, technical justifications for adjusters, accessible language for customers, and formal documentation for regulatory contexts, while multilingual capabilities allow delivery in the customer’s preferred language without a separate translation pipeline.

Finally, multi-step reasoning enables complex calculations by decomposing tasks into sequential operations. Reimbursement assessment may require retrieving the claimed repair cost, identifying per-incident or annual coverage limits, applying absolute or percentage-based deductibles, calculating co-payment where applicable, checking remaining coverage relative to out-of-pocket maximums, and aggregating across multiple line items. In medical contexts, it may also involve verifying that a procedure is covered, confirming whether pre-authorization was obtained, and adjusting reimbursement percentages accordingly. These workflows benefit from stepwise reasoning methods (e.g., approaches described by Wei et al., 2022 [13]), in which intermediate steps can be articulated to support verification and debugging of the calculation logic before the final outcome is presented.

### 2.3.5 Limitations and Challenges

Despite their impressive performance, Large Language Models (LLMs) exhibit several limitations that constrain their deployment in production insurance systems. Understanding these challenges is essential for guiding architectural choices and designing appropriate risk-mitigation strategies.

Hallucinations remain one of the most critical concerns. LLMs can generate content that is syntactically correct and semantically plausible yet factually inaccurate, often with high confidence. In insurance contexts, this may take the form of fabricated policy clauses, incorrect coverage interpretations, or entirely invented customer data. For instance, when queried about flood coverage, a model might assert that “your policy includes flood protection up to 50,000 euros” even if the actual policy excludes flood damage altogether. The persuasive plausibility of such outputs makes them particularly dangerous, as human reviewers may trust authoritative-sounding responses without verification. Empirical studies show that hallucination rates vary across models and tasks, with knowledge-intensive queries producing more errors than reasoning-oriented ones [33] [34]. Effective mitigation requires grounding techniques such as Retrieval-Augmented Generation (RAG), which anchors responses in retrieved source documents, as well as confidence calibration mechanisms to flag uncertain outputs and human oversight for high-stakes cases.

A lack of true understanding also poses fundamental challenges. LLMs are statistical pattern learners rather than agents with genuine semantic comprehension [35]. They predict likely token sequences based on statistical regularities in training data rather than by reasoning about meaning. This results in brittle reasoning behavior: a model may correctly compute reimbursements for hundreds of claims yet fail unexpectedly on a structurally similar case because it lacks the conceptual understanding that human experts rely on. It might accurately interpret "coverage applies unless" constructions in familiar examples but misread semantically equivalent phrasing such as "coverage is void except when." Correct outputs often arise from coincidental correlations rather than from robust inferential logic, producing unpredictable failure modes that can be difficult to anticipate during testing.

The training data cutoff introduces another limitation by constraining temporal knowledge. LLMs only reflect information available up to their last training date, which may precede deployment by months or years. Consequently, policy updates, new regulatory requirements, or coverage changes introduced after that point remain unknown to the model. A model trained before 2023, for example, would not account for pandemic-related policy modifications or recent legal precedents affecting claim interpretation. This limitation underscores the need for architectures such as RAG, which retrieve up-to-date policy documents dynamically rather than relying solely on static parametric knowledge.

Computational cost presents a practical constraint that directly impacts scalability. Running large models incurs substantial API costs and energy consumption [36]. GPT-4 inference is approximately 10 to 100 times more expensive per token than GPT-3.5-turbo, raising cost concerns for high-volume insurance operations.

Processing a single complex claim may involve multiple model calls, for document analysis, information extraction, policy reasoning, and explanation generation, causing cumulative expense. Latency further compounds the issue: inference times range from a few seconds to tens of seconds depending on model size and prompt length, potentially limiting suitability for real-time customer interactions that demand sub-second responsiveness. These constraints motivate careful model selection strategies, using smaller models such as GPT-3.5-turbo for routine extraction tasks while reserving more powerful models like GPT-4 for complex reasoning requiring maximal accuracy.

Persistent bias and fairness concerns further complicate deployment. LLMs inherit statistical and social biases present in their training data, which can inadvertently influence claim assessments [37]. In insurance settings, such biases may manifest as differential treatment linked to demographic attributes: a model might unconsciously associate certain names, geographic regions, or linguistic patterns with higher fraud risk, leading to inequitable outcomes. Gender bias could affect injury-severity assessments, while regional bias might distort cost estimates. Moreover, the predominance of data from specific languages or jurisdictions produces representational gaps that degrade performance for underrepresented populations. Mitigating these effects requires proactive bias testing across demographic groups, continuous fairness monitoring in production, and architectural safeguards such as the exclusion of protected attributes from prompts.

Finally, interpretability challenges pose significant barriers to regulatory compliance and customer trust. The internal decision processes of LLMs are opaque, as knowledge is distributed across billions of parameters in ways that defy direct inspection [38]. When a claim is denied, both customers and regulators expect transparent justification grounded in specific policy clauses. A generic statement such as “claim denied due to policy exclusions” fails to meet legal and ethical standards for explainability. This opacity conflicts with regulations that mandate auditable decision trails and clear reasoning. Although attention visualizations offer limited insight, their correlation with causal decision factors remains weak. More effective approaches involve prompting the model to produce explicit explanations that cite retrieved documents, enabling human reviewers to trace reasoning chains and verify that outputs are grounded in legitimate sources even when the underlying model mechanisms remain inscrutable.

## 2.4 Retrieval Augmented Generation (RAG)

### 2.4.1 Motivation and Architecture

Retrieval-Augmented Generation (RAG) addresses several critical limitations of Large Language Models, particularly hallucination and knowledge cutoff, by grounding language generation in documents retrieved from external knowledge bases. Originally introduced by Lewis et al. (2020) in “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”, the paradigm integrates information retrieval with neural text generation and demonstrated substantial improvements on knowledge-intensive tasks such as open-domain question answering. Subsequent refinements have extended the original architecture to include multi-hop retrieval, iterative refinement, and hybrid retrieval strategies that combine sparse and dense representations [39] [40].

The fundamental motivation behind RAG arises from well-documented deficiencies in parametric knowledge stored within LLM weights. Such knowledge is static, frozen at the time of training, making models unaware of newly introduced facts, updated policies, or regulatory changes. Furthermore, rare or domain-specific information is often underrepresented in training corpora, leading to unreliable recall for specialized queries. Finally, parametric models lack the ability to attribute their answers to verifiable sources, impeding transparency, explainability, and user trust. RAG mitigates these deficiencies by reframing the LLM as a reasoning engine rather than a knowledge repository, dynamically retrieving relevant context from a curated knowledge base and conditioning the generation process on this retrieved information.

Architecturally, RAG augments each input query with relevant retrieved documents prior to generation. Instead of relying exclusively on the knowledge implicitly encoded in its parameters, the model consults an external repository to retrieve semantically relevant passages, which are then concatenated with the query as contextual input. This mechanism ensures that generated outputs are grounded in verifiable sources. The decoupling of knowledge storage, managed by the retrieval system, from reasoning capability, handled by the language model, provides a flexible and maintainable design: updating domain knowledge merely requires refreshing the document index, avoiding the computational and operational burden of model retraining.

A typical RAG system consists of three interdependent components. The Knowledge Base contains domain-specific documents such as policy texts, coverage conditions, regulatory guidelines, and historical claim precedents, preprocessed and segmented into manageable units for efficient indexing. The Retriever identifies the most relevant passages given a user query, using dense vector embeddings to compute semantic similarity and match meaning rather than surface-level lexical overlap. Finally, the Generator, generally implemented as a large language model, receives both the original query and the retrieved documents as input, synthesizing responses grounded in this context. During inference, the generator attends jointly to the query and the retrieved passages, integrating information from multiple sources to construct coherent, contextually justified answers.

This architectural design confers several operational advantages in the insurance domain. Policy updates propagate instantly through the system, as refreshing the knowledge base is sufficient to reflect new information without retraining. Generated answers can include explicit source citations, enabling human reviewers to trace reasoning chains and verify decision logic. The framework also supports heterogeneous document types, including contracts, regulatory circulars, case law, and customer correspondence, stored in a unified retrieval index and accessed through the same semantic search interface. As a result, domain adaptation becomes primarily a data engineering challenge rather than a model training problem, substantially simplifying maintenance while enhancing accuracy, transparency, and compliance.

### 2.4.2 Retrieval Process

Contemporary retrievers encode queries and documents into dense vector embeddings via neural networks, in a process termed Dense Retrieval [41]. This approach fundamentally differs from traditional sparse retrieval methods, representing text as continuous vectors in a high-dimensional space, typically between 768 and 1536 dimensions, where semantic similarity corresponds to geometric proximity. The retrieval process consists of two phases: Offline Indexing, where all documents are embedded and stored, and Online Retrieval, where the query is embedded and compared against the document index.

Similarity between query and document embeddings is computed using cosine similarity or dot product, expressed as:

$$\text{Similarity}(\text{query}, \text{document}) = \cos(\theta) = \frac{\mathbf{e}_q \cdot \mathbf{e}_d}{\|\mathbf{e}_q\|\|\mathbf{e}_d\|} \tag{2.3}$$

Cosine similarity normalizes vectors to unit length before computing the dot product, making it invariant to magnitude and focusing purely on directional alignment in embedding space.

Values range from -1, representing opposite directions, to 1, indicating identical directions, with typical relevant documents scoring between 0.6 and 0.9 depending on specificity.

The retriever computes similarity between the query embedding and all document embeddings, ranks documents by descending similarity, and returns the top-k results, typically between 3 and 10 for insurance applications, balancing recall against context window constraints.

Dense retrieval captures semantic relationships that keyword-based methods such as BM25 [42] often miss. BM25 ranks documents based on term frequency and inverse document frequency, rewarding documents containing query words while penalizing common terms. However, this approach fails when queries and documents use different vocabularies to express the same concept. A query about “broken windshield” might not retrieve a policy document discussing “automotive glass damage” because there is no lexical overlap beyond “damage”. Dense retrieval, by encoding meaning rather than words, recognizes semantic equivalence and retrieves the relevant policy clause. This proves particularly valuable for multilingual scenarios, where a query like “cristalli rotti” (Italian for “broken glass”) can successfully retrieve English policy documents because embeddings capture cross-lingual semantic similarity when trained on multilingual corpora.

The choice of embedding model significantly impacts retrieval quality. Sentence-BERT (SBERT) [43] fine-tunes BERT using siamese networks to produce semantically meaningful sentence embeddings. Unlike standard BERT, which requires processing sentence pairs jointly and is computationally expensive at retrieval time, SBERT embeds sentences independently, enabling efficient similarity search. The model trains on sentence pair datasets with labels indicating semantic equivalence, learning to place similar sentences close together in embedding space. OpenAI's text-embedding-ada-002 [44] and more recent embedding families with comparable dimensionality and cost profiles represent proprietary models balancing quality, efficiency, and cost. These models produce 1536-dimensional embeddings and demonstrate strong performance across diverse domains without fine-tuning, making them suitable for production deployments where training infrastructure may be limited.

Domain-specific embeddings fine-tuned on insurance corpora can further improve retrieval quality for specialized terminology [45]. Insurance documents contain technical terms such as "subrogation", "underwriting", "deductible", and "rider", which may not appear frequently in general-purpose training data. Fine-tuning on domain-specific corpora ensures that these terms receive appropriate embeddings reflecting their specialized meanings. This process typically involves continued training on sentence pairs drawn from insurance documents, teaching the model that "comprehensive coverage" and "collision coverage" are distinct concepts despite lexical similarity, and that "premium" in insurance contexts refers to payment rather than quality.

### 2.4.3 Vector Databases

Efficient large-scale retrieval requires specialized data structures optimized for high-dimensional vector search. Vector databases are systems designed specifically to store and query dense embeddings, addressing the computational challenge of finding nearest neighbors in spaces with hundreds or thousands of dimensions. Unlike relational databases that support exact matches through indexing using B-trees or hash tables, vector databases implement Approximate Nearest Neighbor (ANN) search, trading perfect accuracy for orders of magnitude speedup. For a database with N documents, exhaustive search requires computing N similarity scores, resulting in O(N) complexity, which becomes prohibitively expensive as collections grow to millions of documents. ANN algorithms reduce this to logarithmic or sub-linear complexity while maintaining high recall, typically retrieving 95 to 99 percent of true nearest neighbors.

HNSW, or Hierarchical Navigable Small World, represents a graph-based algorithm that provides fast ANN search with high recall [46]. The data structure builds a multi-layer proximity graph where each layer contains nodes representing document embeddings connected to their nearest neighbors. Upper layers contain sparse subsets with long-range connections enabling rapid navigation toward the query region, while lower layers contain progressively denser graphs refining the search. During query time, the algorithm starts at the top layer, traverses edges greedily toward the query vector, and descends through layers until reaching the bottom, where it performs local search to identify the nearest neighbors. This hierarchical structure achieves logarithmic search complexity $O(\log N)$ while maintaining recall above 95 percent in practice. Construction time is $O(N \log N)$, with tunable parameters controlling the number of edges per node and balancing memory usage against search speed.

FAISS, or Facebook AI Similarity Search [47], offers a comprehensive library implementing various indexing methods optimized for different trade-offs among speed, memory, and accuracy. The library includes flat indexes for exhaustive search, product quantization for memory compression, IVF indexes for partitioned search, and GPU-accelerated implementations for high-throughput scenarios. Product quantization decomposes vectors into subvectors and quantizes each independently, reducing memory requirements by 10x to 50x while maintaining acceptable recall. This compression proves valuable when deploying on memory-constrained infrastructure or when the embedding collection exceeds available RAM.

IVF, or Inverted File Index [48], partitions the vector space into clusters using k-means or similar algorithms, then searches only relevant partitions at query time. During indexing, all vectors are clustered into C partitions, typically C equals $\sqrt{N}$, and each vector is assigned to its nearest cluster centroid. At query time, the algorithm identifies the nearest cluster centroids to the query vector and searches only within those clusters, examining a small fraction of the database. This reduces search complexity from $O(N)$ to $O\left(\frac{N}{C} + C\right)$, with the optimal partition count balancing exhaustive centroid comparison against within-cluster search. IVF scales effectively to billions of vectors when combined with quantization techniques.

Among vector database systems, ChromaDB, Pinecone, Weaviate, and Milvus represent the most popular production-ready solutions, each with distinct architectural characteristics. ChromaDB emphasizes simplicity and local deployment, offering an embedded Python library that requires no separate server process, making it suitable for development environments and small-scale production use. Pinecone provides a fully managed cloud service optimized for scalability and operational simplicity, handling indexing, replication, and serving infrastructure transparently. Weaviate supports hybrid search combining vector similarity with keyword filtering and graph relationships, enabling complex queries such as “find similar policies for customers in region X with premium above Y euros.” Milvus targets high-throughput enterprise scenarios with a distributed architecture supporting horizontal scaling across clusters.

Selection among these systems depends on factors such as the need for local versus cloud deployment, expected query volume, latency constraints, integration with existing infrastructure, and operational complexity tolerance. For insurance applications with moderate document collections, typically ranging from thousands to millions of policies, embedded solutions like ChromaDB often suffice, whereas high-volume customer-facing platforms may benefit from managed cloud services such as Pinecone or Weaviate.

## 2.4.4 RAG Workflow

A production Retrieval-Augmented Generation workflow consists of several coordinated phases, each addressing specific technical challenges and contributing to the overall system reliability.

Query processing begins with the user question being embedded into a vector representation using the same embedding model employed during document indexing. This consistency ensures that queries and documents inhabit the same embedding space, making similarity scores meaningful. Some systems also apply preprocessing steps such as spelling correction, abbreviation expansion, or query reformulation to improve retrieval quality. In insurance applications, for example, preprocessing may normalize policy identifiers by removing hyphens or standardizing case, and expand domain-specific acronyms such as “RCA” into “Responsabilità Civile Auto”. Advanced implementations perform query expansion, where the original query generates semantically related variations to enhance recall; a query about “windshield damage” might expand to include “glass repair” and “automotive glazing”.

The retriever then compares the query embedding against all document embeddings stored in the vector database using the Approximate Nearest Neighbor algorithms described previously. The top-k most similar documents are retrieved, with k typically ranging from three to ten depending on the available context window and the desired specificity. This phase involves critical trade-offs: retrieving too few documents risks low recall and potential information gaps, while retrieving too many may dilute the context with marginally relevant material that confuses the generator. Systems therefore balance precision and recall carefully. Some implementations use dynamic k selection based on similarity thresholds, retrieving all documents above a similarity cutoff rather than a fixed number. This adaptive approach proves particularly effective when query specificity varies; narrow queries such as “what is the deductible for policy 12345?” retrieve a small number of high-confidence matches, while broader queries such as “what damages are covered?” collect more documents spanning multiple coverage categories.

The next step involves context construction, where retrieved documents are formatted and concatenated with the user query to form an augmented prompt for the generator. During this phase, several engineering decisions affect generation quality. Documents are generally ordered by descending similarity score, although some systems randomize order to mitigate position bias, where the model tends to overemphasize earlier context. Citation generation is facilitated by including metadata for each document chunk, such as source identifier, chunk position, and retrieval score. The prompt template structures the context, typically following patterns such as: “Answer the following question based on the provided policy documents. Question: [query]. Policy Documents: [doc1], [doc2], [doc3]. Answer:". More sophisticated templates introduce explicit instructions for uncertainty handling, such as "if the documents do not contain sufficient information, say so explicitly", and citation requirements like "cite the specific policy clauses supporting your answer".

Generation follows by feeding the augmented prompt to the Large Language Model, which processes both the query and retrieved documents, attending over all tokens to synthesize an answer grounded in the provided context. The model's attention mechanism naturally focuses on the most relevant passages, though this weighting emerges implicitly rather than through explicit programming. Output determinism and creativity are governed by temperature and sampling parameters: lower temperatures between 0.0 and 0.3 yield deterministic factual responses suitable for insurance queries, while higher temperatures between 0.7 and 1.0 encourage creativity for open-ended tasks. Maximum token limits constrain response length, with typical insurance-related answers ranging from 100 to 500 tokens depending on query complexity.

Post-processing validates, formats, and refines the generated response. Validation steps ensure internal consistency and factual correctness by checking that cited policy numbers exist in the database, that calculated reimbursement amounts align with deterministic computation modules, and that the response avoids prohibited content such as speculative advice or unauthorized coverage claims. Citation extraction identifies referenced documents in the generated text, enabling the creation of hyperlinks or footnotes for human verification. Some systems also implement confidence scoring, prompting the model to rate its certainty or estimating confidence from token probabilities, with low-confidence outputs flagged for human review.

Formatting adapts the final output to the target interface, producing structured JSON for APIs, formatted HTML for web dashboards, or fluent natural language for conversational chatbots.

Finally, robust error handling ensures that failure modes are managed gracefully. When retrieval returns no relevant documents, the system defaults to reporting insufficient information. If generation exceeds token limits, truncation or summarization safeguards the process. In cases of API timeout or transient network errors, retry mechanisms with exponential backoff maintain continuity without overloading the system.

### 2.4.5 Advantages for Insurance Applications

RAG systems offer compelling advantages for insurance claims processing, effectively addressing both technical and business requirements specific to the domain.

Accuracy and grounding improve substantially because RAG systems explicitly provide policy text as contextual input, anchoring generation in verifiable sources. When a customer asks, “Does my policy cover hail damage?”, the system retrieves the specific policy clauses addressing weather-related coverage and generates an answer based on those retrieved passages rather than relying on the model’s internal parametric knowledge. This grounding significantly reduces hallucinations, since the model cannot fabricate coverage terms when it must reference the actual policy text provided in the prompt. Empirical research demonstrates that RAG reduces factual errors by 30 to 50 percent compared to pure parametric generation on knowledge-intensive tasks [16] [17]. In insurance applications, where incorrect coverage interpretations carry potential legal and financial consequences, this improvement is of critical importance.

Transparency and explainability also benefit from RAG’s architecture because the retrieved documents serve as explicit evidence that enables human reviewers to verify correctness. Unlike pure LLM-generated responses, where the reasoning process remains opaque, RAG systems can cite specific policy sections supporting each statement, such as “Your windshield replacement is covered under Comprehensive Coverage (Section 4.2.b) with a 150 euros deductible (Schedule A, Line 7).” Both adjusters and customers can trace these citations back to their sources, verifying the model’s interpretation directly. This design satisfies regulatory explainability requirements established by GDPR and the EU Insurance Distribution Directive, which mandate that insurers explain automated decisions affecting consumers. The ability to trace responses to specific policy clauses further supports audit trails, ensuring compliance and facilitating quality assurance reviews.

Information currency remains high because policy updates propagate instantly through the knowledge base without requiring model retraining. When an insurer introduces a new coverage type, updates premium calculations, or modifies exclusions in response to regulatory changes, these updates involve only adding or modifying documents in the vector database and re-embedding them. The retrieval and generation components continue to operate without modification. This agility is particularly valuable in insurance, where policy terms evolve frequently, for example due to regulatory adjustments following natural disasters, pandemic-related coverage revisions, or competitive pressures driving new product designs. In contrast, fine-tuned models require retraining cycles that can last days or weeks, during which the deployed system may continue to provide outdated information. By contrast, RAG systems update within minutes, maintaining full alignment between the knowledge base and customer-facing responses.

In this paradigm, domain adaptation becomes primarily a data engineering challenge rather than an expensive model training task. RAG leverages general-purpose LLMs by grounding them in company-specific documentation, thereby avoiding domain-specific fine-tuning that demands labeled datasets, GPU infrastructure, and machine learning expertise. An insurer can deploy a RAG system simply by curating its policy documents into a structured knowledge base and configuring retrieval parameters, tasks that remain accessible to engineering teams without specialized AI training. This democratization of AI deployment lowers the barrier to adoption significantly. Moreover, a single LLM can serve multiple insurance lines, such as auto, home, life, and health, by maintaining separate document collections for each domain, whereas fine-tuned approaches would require distinct models for each.

Scalability likewise improves due to the filtering nature of retrieval, which ensures that only the most relevant sections of extensive policy documentation are passed to the LLM. Insurance companies typically maintain vast policy libraries containing hundreds of variants across product lines, regions, regulatory contexts, and historical versions. For example, a comprehensive auto insurance policy corpus may span thousands of pages, far exceeding the capacity of even large-context LLMs. RAG mitigates this limitation by retrieving only the most relevant passages for each query. A question about comprehensive coverage, for instance, retrieves only those relevant sections rather than loading the entire policy as context.

This selective retrieval enables efficient processing of knowledge bases containing millions of tokens using models with context windows limited to tens of thousands of tokens, achieving logarithmic rather than linear scaling with respect to corpus size.

### 2.4.6 Challenges and Advanced RAG Techniques

Despite their advantages, RAG systems face several fundamental challenges that constrain performance and continue to motivate research into advanced retrieval and generation techniques.

Retrieval quality represents the most critical factor affecting system accuracy because if the retriever fails to identify relevant documents, the generator cannot produce correct answers regardless of its reasoning capability. Retrieval failures manifest in several forms: the embedding model may fail to capture semantic similarity for domain-specific terminology, the query phrasing may differ significantly from document phrasing despite semantic equivalence, or relevant information may be distributed across multiple chunks, none of which individually appears strongly relevant. For example, determining coverage for a particular claim scenario may require combining information from a general coverage clause, an exclusions list, and a special conditions addendum, yet each chunk on its own receives only a moderate relevance score. Improving retrieval quality depends on multiple factors, including high-quality embeddings tuned for insurance terminology, careful parameter calibration balancing precision and recall, and hybrid retrieval approaches that combine vector similarity with keyword-based matching to ensure comprehensive coverage.

Context length limitations pose another significant challenge. Even with extended context windows, practical limits constrain how much retrieved information can be included in a single generation prompt. For instance, GPT-4 variants support up to 32,000 tokens, but effective context utilization degrades as the prompt grows longer. The "lost in the middle" phenomenon [49] shows that models tend to attend more to information at the beginning and end of the prompt, under-weighting content appearing in the middle. This issue becomes particularly problematic for complex insurance policies spanning hundreds of pages, where even aggressive retrieval filtering may produce dozens of relevant chunks totaling tens of thousands of tokens. Selecting which information to include therefore requires a balance between comprehensiveness and the constraints of the model's context window and attention mechanisms.

Chunking strategy selection also critically influences retrieval effectiveness. Documents must be divided into chunks before embedding, and chunk size introduces inherent trade-offs. Smaller chunks containing approximately 100 to 200 tokens offer precise retrieval by aligning narrowly focused passages with user queries but lose the surrounding context necessary for correct interpretation. Larger chunks containing 500 to 1000 tokens preserve context but yield less precise retrieval, often including irrelevant material alongside the relevant portion. Achieving optimal performance requires balancing these opposing tendencies. Chunking should avoid severing related information, since splitting mid-sentence produces incoherent fragments while separating closely related paragraphs disconnects context from content. Advanced strategies mitigate these issues by employing sliding windows with overlap, typically maintaining a 50 to 100 token overlap between consecutive chunks to preserve cross-boundary information. Others use semantic segmentation, leveraging natural language processing to detect paragraph boundaries, section headers, or topic shifts using sentence embeddings, thereby ensuring that each chunk represents a coherent semantic unit.

Metadata filtering introduces another layer of complexity because insurance queries often include contextual constraints beyond pure semantic similarity. A query about "deductible amounts," for instance, should retrieve information only from the customer's active policies, excluding expired contracts or documents belonging to other customers. Achieving this requires combining vector similarity with structured metadata predicates such as customer ID, policy type, effective date, or jurisdiction. Implementations typically adopt a two-stage approach: first, metadata constraints narrow the search space, and second, vector similarity search operates within that filtered subset. More sophisticated systems implement compound indexing structures that support simultaneous evaluation of metadata filters and vector similarity, though these solutions substantially increase implementation complexity.

Recent innovations propose advanced methods to address these limitations. One such method is Hypothetical Document Embeddings (HyDE) (Gao et al., 2022), which reverses the traditional retrieval order. Instead of embedding the query directly, the system first prompts the LLM to generate a hypothetical answer to the query, then embeds that answer for retrieval. The intuition is that a generated answer often aligns more closely with the phrasing and structure of actual documents, thereby improving retrieval recall. For insurance applications, a query such as “Am I covered for flood damage?” might yield a hypothetical answer like “Yes, flood damage is covered under comprehensive policies with specific exclusions for...,” which semantically resembles real policy language more than the brief initial query.

Another technique, multi-query retrieval, reformulates the original query into multiple semantically related variants and performs retrieval for each. For example, a single query about “windshield repair” may be expanded into “automotive glass damage,” “windscreen replacement,” and “front window coverage,” each retrieving distinct yet relevant documents reflecting terminology variation across insurers or product lines. The union of retrieved documents provides more comprehensive context than any single query variant, significantly improving recall in heterogeneous policy corpora.

A further improvement involves re-ranking strategies, which refine the ordering of retrieved documents before generation [50]. The retrieval process begins with fast but less accurate methods, such as dense embedding search, to identify a broad candidate set. A more computationally intensive cross-encoder model then processes each query–document pair jointly to produce highly accurate relevance scores. Cross-encoders outperform bi-encoders because they model token-level interactions between queries and documents, but their computational cost prevents their use over the full corpus. The two-stage design reconciles this trade-off by retrieving the top 100 candidates efficiently and re-ranking the top 10 with high accuracy, combining scalability with precision.

Recursive retrieval introduces iterative reasoning by retrieving, generating partial answers, and then retrieving again based on intermediate results (Khattab et al., 2021). For complex multi-hop queries, the system first retrieves documents related to the initial question, generates partial answers or identifies missing information, formulates refined follow-up queries, and retrieves additional supporting documents. For instance, determining whether a specific claim is covered may require first retrieving the customer’s policy type, then retrieving the coverage terms applicable to that policy, and finally identifying exclusions that might apply. This iterative refinement allows RAG systems to handle queries that require integrating information across multiple documents and reasoning steps, closely mirroring the workflow of human adjusters.

## 2.5 Optical Character Recognition (OCR) and Document Processing

### 2.5.1 OCR Technology Evolution

The Optical Character Recognition (OCR) technology has undergone substantial evolution over the past five decades, transitioning from rigid template-based systems to flexible deep learning architectures. This evolution has had direct implications for insurance claims processing, where document quality and format variability demand highly robust extraction capabilities.

Traditional OCR systems developed between the 1970s and early 2000s relied on template matching and handcrafted feature extractors [2]. These systems encoded explicit rules describing character shapes, strokes, and patterns. Their performance was acceptable for clean, high-resolution printed text with standard fonts, but degraded substantially when confronted with handwritten content, poor scan quality, varied typefaces, or complex document layouts, all of which are common in insurance claims documentation. Moreover, template-based approaches required extensive manual engineering for each new font or language, limiting scalability and generalization. Error rates for handwritten text frequently exceeded 30 to 40 percent, rendering such systems impractical for claims forms containing handwritten sections.

The introduction of Tesseract OCR represented a pivotal milestone in open-source OCR technology. Originally developed by Hewlett-Packard in the mid-1980s as a proprietary system, it was released as open source by Google in 2005 [51] [7]. Tesseract quickly became the de facto standard for open-source OCR due to its maturity, broad language support exceeding one hundred languages, and competitive accuracy on printed text. Early versions (1.x–3.x) relied on classical computer vision methods such as connected component analysis for character segmentation, feature extraction based on character topology, and pattern matching against predefined character templates.

The fourth generation, Tesseract 4.0, released in 2018, integrated LSTM-based neural networks (Graves et al., 2009), marking a major architectural shift. Instead of segmenting characters individually and recognizing them in isolation, the LSTM recognizer processes entire text lines as sequences, enabling the model to exploit contextual information. For example, in the sequence "ins_rance policy", the LSTM can infer that the missing character is "u" based on surrounding context, whereas isolated character recognition would fail.

This context-aware approach substantially improved robustness, especially for degraded documents, unusual fonts, and connected scripts. As a result, Tesseract 4.x achieves character error rates of 1–5 percent on clean printed documents and 5–15 percent on moderately degraded scans, compared to 5–20 percent for earlier versions on similar material.

During the 2010s, deep learning-based OCR emerged, introducing end-to-end architectures that eliminated manual feature engineering entirely. These systems combined Convolutional Neural Networks (CNNs) for visual feature extraction with Recurrent Neural Networks (RNNs) or Transformers for sequence modeling [52]. The objective was to integrate the strengths of both paradigms: CNNs learn to detect edges, curves, strokes, and higher-level character structures directly from pixel data without handcrafted features, while RNNs or LSTMs process these visual feature sequences, predicting character strings with awareness of linguistic context and structure. This architecture enabled large-scale training on millions of document images, producing robust models that generalize across fonts, languages, and diverse document conditions.

Commercial cloud OCR services, including Google Cloud Vision API, AWS Textract, and Azure Read API, have since adopted proprietary deep learning models trained on massive internal datasets. These services typically outperform open-source alternatives like Tesseract, especially on challenging inputs such as handwriting, degraded scans, or complex layouts, achieving character error rates as low as 0.5–2 percent for printed text and 10–20 percent for handwriting. However, these systems incur recurring costs of approximately 1.50 USD per 1,000 pages, which can become prohibitive for high-volume insurance operations. For instance, an insurer processing 100,000 claim documents annually would spend between 15,000 and 30,000 USD solely on OCR, while Tesseract, being open-source and locally executable, incurs only infrastructure costs such as compute time, typically negligible compared to API fees.

The current frontier in document understanding is represented by multimodal models, which integrate textual, spatial, and visual reasoning. Notable examples include LayoutLM [53] and its successors LayoutLMv2 and LayoutLMv3 [54]. These models are pre-trained on millions of document images, learning to jointly represent text content, spatial layout, including the positions of text blocks, tables, and figures, and visual attributes such as fonts, colors, and formatting. This multimodal representation enables holistic document comprehension, allowing the model to recognize that a number appearing in a specific table cell corresponds to a particular coverage type based on the associated row and column headers, or that text in a header section defines the context for subsequent content.

Another breakthrough model, Donut [55], adopts an OCR-free approach by using a vision-to-text Transformer that directly processes document images and outputs structured information, typically in JSON format, without explicit character recognition. This design bypasses traditional OCR pipelines entirely, offering a more integrated alternative to conventional text extraction and parsing.

For insurance applications, these advanced multimodal models enable the extraction of structured data from highly complex policy and claim documents. They are particularly effective for cases involving premium tables with conditional pricing, coverage grids with multiple dimensions, or claim forms that include checkboxes indicating coverage types, handwritten damage descriptions, and mixed-format receipts. Such models achieve higher accuracy with significantly less engineering effort compared to traditional OCR-plus-parsing pipelines, making them especially valuable for modern insurance claim automation systems.

### 2.5.2 OCR Challenges in Insurance Documents

Insurance claims involve heterogeneous document types that present distinct OCR challenges, each of which must be addressed to achieve a robust and reliable document processing pipeline.

Scanned policy documents frequently arrive as low-quality images exhibiting multiple forms of degradation. Repeated photocopying or faxing introduces both random and systematic distortions: noise and speckles appear as random pixel variations or dust marks, while text edges become fuzzy or broken, often accompanied by JPEG compression artifacts and moiré patterns resulting from scanning printed halftones. Additional quality issues arise when pages are scanned at angles, producing geometric skew that requires rotation correction, or when documents exhibit uneven illumination, leaving some regions underexposed or washed out. In insurance, where many policy documents may be decades old and have been copied or faxed numerous times, such low-quality inputs are the norm rather than exceptions.

Handwritten claim forms present even greater challenges because handwriting varies substantially across individuals in terms of character shape, slant, spacing, and stroke continuity. Some writers produce well-separated printed characters, while others use cursive script with tightly connected strokes and ambiguous boundaries between characters. As a result, handwriting recognition typically requires specialized models trained on handwritten datasets that differ from models used for printed text. The gap in performance is significant: character error rates for handwriting range between 15 and 40 percent depending on writing quality, compared to 1 to 5 percent for clean printed text. In insurance claims, critical fields such as policy numbers, claim amounts, and dates are often handwritten, which necessitates rigorous OCR quality validation and human verification whenever low-confidence extractions occur.

Mixed-content documents further complicate processing by combining diverse visual elements such as text, tables, checkboxes, signatures, stamps, and photographs on a single page. Traditional OCR engines treat the entire page as uniform text, an assumption that fails in these heterogeneous contexts. Checkboxes require the detection of visual marks such as “,” “,” or filled boxes rather than textual characters, and tables demand the preservation of row and column structures to maintain logical relationships. Naïve OCR typically returns text in linear reading order, losing the association between table cells and headers. Signatures and stamps, which are graphical elements rather than text, may be misinterpreted as nonsensical characters, while embedded photographs or diagrams should either be ignored or handled by separate processing pipelines. For insurance claim forms containing structured sections, including checkboxes indicating coverage types, tables listing itemized damages, free-text fields for incident descriptions, and designated areas for signatures, accurately distinguishing and processing each content type is a non-trivial requirement.

Multilingual documents introduce another layer of complexity, particularly in multinational insurance contexts where a single claim form might contain Italian instructions, English policy references, and handwritten notes in a regional dialect. OCR engines generally perform best when the target language is specified a priori, yet many insurance workflows must handle mixed-language inputs dynamically. This necessitates automated language detection at the page, region, or line level, followed by adaptive switching of OCR models or configurations. While multilingual OCR models exist, they often trade off accuracy compared to language-specific systems. For insurers operating in multilingual regions such as Switzerland (German, French, Italian), Belgium (French, Dutch), or Canada (English, French), effective multilingual OCR is indispensable rather than optional.

Complex document layouts pose additional difficulties. Insurance documents frequently feature multi-column text, nested tables, non-linear reading orders such as sidebars or footnotes, and embedded figures. Traditional OCR engines output text in a simple left-to-right, top-to-bottom sequence, which leads to misordered results in multi-column formats, for example, reading "Column 1 Line 1, Column 2 Line 1, Column 1 Line 2, Column 2 Line 2" instead of completing each column sequentially. Nested tables, where tables appear within other cells, challenge most layout analysis algorithms, often producing incomplete or distorted extractions. In insurance policy documents containing multi-column coverage descriptions, embedded premium tables, or margin notes referencing specific clauses, preserving structural relationships during extraction is essential for subsequent processing stages such as semantic parsing, clause retrieval, or data entry automation.

### 2.5.3 Image Preprocessing Techniques

Obtaining high-quality OCR results depends critically on the preprocessing phase, which applies a series of image enhancement and correction steps before recognition. During this phase, raw scanned images are transformed into forms optimized for character recognition, often improving OCR accuracy by 10 to 30 percent on degraded documents.

Binarization converts grayscale or color images into binary form, typically representing text as black on a white background or vice versa. This simplification enhances contrast and separation between text and background, improving the recognizer's ability to distinguish characters. Global thresholding applies a single threshold value across the entire image: pixels darker than the threshold become black, while lighter pixels become white. Although simple and computationally efficient, this method fails for documents with uneven illumination, where some regions are underexposed or overexposed. Otsu's method [56] mitigates this limitation by automatically determining the optimal threshold that maximizes inter-class variance between foreground and background, but it still performs best when the image has a bimodal intensity distribution, meaning a clear distinction between text and background. Adaptive thresholding, by contrast, computes thresholds locally for small image regions, effectively handling variable lighting. The Sauvola method [57] achieves particularly strong results for degraded documents because it adjusts thresholds dynamically based on local mean and standard deviation. For insurance documents scanned under inconsistent lighting conditions, adaptive methods typically outperform global thresholding approaches.

Noise reduction is another crucial preprocessing step that removes random pixel variations, speckles, dust marks, and compression artifacts which otherwise interfere with OCR accuracy. Median filtering replaces each pixel with the median of its neighbors, effectively removing salt-and-pepper noise, isolated dark or light pixels, while preserving edge details. Gaussian filtering smooths noise through weighted averaging but can blur character edges if parameters are not tuned carefully. Bilateral filtering [58] offers a more balanced approach, smoothing noise while preserving edges by weighting neighboring pixels based on both spatial distance and intensity similarity. Morphological operations such as erosion, dilation, opening, and closing can further remove small noise artifacts or repair gaps in broken characters. For insurance claims involving photocopied or faxed documents, where artifacts and texture noise are frequent, these techniques are essential to achieve acceptable OCR accuracy.

Deskewing corrects geometric misalignment introduced when documents are scanned at slight angles. Even small skew angles of 2 to 3 degrees can reduce OCR accuracy by 5 to 10 percent because recognition models assume horizontally aligned text lines. Skew detection algorithms estimate the dominant orientation of text lines using the Hough transform, projection profiles, or connected component analysis [59]. Once the skew angle is detected, the image is rotated to align the text horizontally. In insurance workflows where many documents are scanned manually, automatic deskewing is indispensable.

Dewarping addresses a different geometric distortion, correcting page curvature in photographs of open books, bound documents, or wrinkled pages. Such distortions arise because the page surface is non-planar, causing curved baselines and warped character shapes. Dewarping algorithms detect curved text lines, estimate the three-dimensional page surface, and apply geometric transformations to flatten it [60]. This technique is especially relevant when customers photograph folded policy booklets or bent claim forms instead of scanning them flatly.

Contrast enhancement improves the visibility of faint or low-contrast text, particularly in aged or faded documents. Histogram equalization redistributes pixel intensities across the full range, increasing global contrast, while adaptive histogram equalization (CLAHE,Contrast Limited Adaptive Histogram Equalization) performs localized adjustments that enhance contrast in text regions without over-amplifying noise. Gamma correction modifies brightness non-linearly, brightening shadows or darkening highlights as needed. For insurance documents with faint print, such as carbon copies or aged policies, these methods can significantly boost OCR performance.

Resolution adjustment ensures images meet the optimal input requirements of the OCR engine. Tesseract performs best at approximately 300 dots per inch (DPI). Images scanned at lower resolutions, such as 72 to 150 DPI, typical for quick smartphone captures, lack sufficient detail for accurate recognition. Upsampling through bilinear or bicubic interpolation, or using deep learning super-resolution models [61], can partially compensate for low resolution, while overly high-resolution images above 600 DPI may include excessive detail such as paper texture and ink irregularities, which can reduce accuracy. In those cases, downsampling improves both recognition and processing efficiency.

In insurance claim processing pipelines, these preprocessing methods are often applied sequentially: binarization enhances contrast, noise reduction removes artifacts, deskewing aligns text lines, dewarping corrects curvature when necessary, contrast enhancement clarifies faint text, and resolution adjustment ensures compatibility with the OCR engine. The optimal preprocessing configuration depends on the characteristics of each document, which often requires adaptive preprocessing pipelines capable of automatically analyzing image quality and selecting the appropriate combination of methods and parameters.

### 2.5.4 Hybrid OCR Strategies

For production insurance claims systems, hybrid approaches that combine multiple extraction methods often yield superior results compared to any single method, since the diversity of document types, ranging from digitally created PDFs to scanned documents and photographs, and quality levels, from pristine to severely degraded, necessitates adaptive strategies capable of selecting optimal extraction methods based on document characteristics.

PDF text extraction provides the foundation for these hybrid strategies. When PDFs contain embedded text, created electronically from word processors, policy management systems, or digital forms rather than scanned, text can be extracted directly using libraries such as PyMuPDF (fitz) or PDFMiner. These libraries parse PDF data structures and extract text together with metadata (fonts, positions, formatting) without requiring any character recognition. Consequently, this approach entirely avoids OCR, achieving near-perfect accuracy for well-formed digital PDFs, as the error rate is effectively zero: the extracted text is exactly what the PDF contains, with no recognition errors. Moreover, native extraction is typically 10x to 100x faster than OCR (milliseconds versus seconds per page), providing significant throughput advantages. In high-volume insurance claims processing, where many policy documents are digitally generated PDFs, native extraction therefore dramatically reduces processing time while improving accuracy.

However, native extraction completely fails for scanned or image-based PDFs, since when a PDF contains only scanned page images rather than embedded text, native extraction returns empty strings or meaningless symbols. As a result, systems must detect the PDF type and route documents to appropriate extraction methods. Detection heuristics can analyze extraction output: very few characters extracted (for instance, fewer than 50 for a full-page document) suggest a scanned PDF, unusual character distributions (many non-alphabetic characters or random symbols) indicate extraction errors, and missing expected content (such as policy numbers, dates, or monetary amounts in known formats) reveals that native extraction has failed.

OCR fallback mechanisms ensure that scanned documents are still processed when native extraction yields sparse or low-quality results. When detection heuristics indicate that a PDF is scanned or image-based, the system automatically switches to OCR processing. This provides robustness, as OCR can handle any document that can be rendered as an image, including scanned PDFs, photographs, or faxes, thus preventing complete failure on legacy or low-quality submissions. Importantly, the system does not require manual configuration or user input regarding document type. Instead, automatic quality assessment of native extraction output triggers OCR fallback only when necessary, thereby optimizing both speed (through native extraction when possible) and accuracy (through OCR when required).

Multi-engine ensemble approaches extend this concept by running multiple OCR engines and combining their outputs. Different OCR engines, such as Tesseract, Google Cloud Vision API, AWS Textract, and Azure Read API, possess complementary strengths based on their training data, model architectures, and optimization targets. For instance, Tesseract excels at printed text in supported languages, Google Vision performs well on handwriting and degraded documents, and AWS Textract specializes in forms and tables through structured field extraction. Consequently, for challenging documents involving handwriting, degraded scans, or unusual layouts, multi-engine voting or confidence-based selection can reduce error rates by 20% to 40% compared to single-engine approaches. Voting methods run multiple engines and select the most common output for each word or field, whereas confidence-based selection chooses the output reported with the highest confidence score.

Nevertheless, multi-engine approaches incur costs, since running N engines multiplies both expenses (in terms of API fees for cloud services) and latency (as serial execution adds up individual engine times, while parallel execution demands N× compute resources). For cloud OCR APIs that charge per page, Google Vision and AWS Textract both approximately \$1.50 per 1000 pages, ensemble approaches become expensive at scale. For example, an insurer processing 100,000 documents annually with a dual-engine ensemble would incur roughly \$300 in OCR costs compared to \$150 for a single engine or near-zero for Tesseract alone. Moreover, adding a third or fourth engine often provides only minimal accuracy gains beyond two engines, reflecting diminishing returns and suggesting that dual-engine ensembles may represent the optimal cost–benefit balance.

In the insurance claims domain, therefore, selective ensemble use is the most prudent strategy. Single-engine OCR, such as Tesseract, can handle routine documents, while multi-engine ensembles are reserved for high-value claims or cases where single-engine confidence is low. This approach maintains high accuracy for critical claims while controlling costs for routine processing.

Human-in-the-loop verification further addresses OCR uncertainty for critical fields. For fields with high business impact, such as policy numbers, claim amounts, coverage limits, or dates, low-confidence OCR results can be flagged for human review. Rather than processing all documents manually or accepting all OCR outputs uncritically, selective human verification directs human effort toward uncertain extractions. For instance, if OCR extracts a policy number "AB12345" with 95% confidence, it can be accepted automatically.

However, if confidence drops to 60%, a human reviewer verifies the extraction against the original image. This hybrid human–AI approach thus balances automation efficiency, as most extractions proceed automatically, with accuracy, as uncertain extractions receive targeted verification.

Finally, quality assessment and confidence scoring underpin all these hybrid strategies. Tesseract and most cloud OCR APIs provide per-word or per-character confidence scores that indicate recognition certainty, and the aggregate confidence (mean word confidence across the extracted text) reflects overall extraction quality. Thresholds, such as triggering human review or multi-engine processing when mean confidence falls below 70%, enable a balanced trade-off between automation and accuracy. Cross-validation further enhances reliability: when multiple extraction methods yield substantially different outputs, this suggests either a problematic PDF (for example, corrupted or unusually encoded) or a failed OCR, warranting human inspection. The effectiveness of a hybrid OCR strategy therefore depends critically on accurate quality assessment, since poor assessment, whether selecting

OCR unnecessarily when native extraction would suffice or trusting low-quality OCR, can entirely negate the benefits of the hybrid approach.

### 2.5.5 Tesseract OCR: Architecture and Usage

Tesseract OCR merits detailed examination as the most widely deployed open-source OCR engine and a common choice for insurance claims processing systems requiring cost-effective, locally executable text extraction.

Tesseract 4.x architecture combines traditional computer vision with modern deep learning, and its processing pipeline proceeds in several stages. Page layout analysis first detects text regions, separating text from non-text areas such as images, white space, or decorative elements. It then identifies text blocks (paragraphs, columns, headers), segments these blocks into lines, and further segments lines into words. This process relies on connected component analysis, grouping connected black pixels into character-like blobs, analyzing blob properties (size, shape, position), and clustering blobs into words and lines based on spacing and alignment. Layout analysis is critical for complex documents, since incorrect segmentation, such as merging separate words, splitting single words, or confusing columns, propagates errors into the recognition stage.

Character recognition employs LSTM neural networks. Unlike earlier Tesseract versions that recognized isolated characters, Tesseract 4.x processes entire text lines as sequences. The LSTM receives a sequence of visual features, extracted from the line image via convolutional layers, and outputs a sequence of characters. This sequence-to-sequence approach enables the model to leverage contextual information, allowing uncertain characters to be resolved based on surrounding letters and linguistic patterns. For example, in the phrase "ins_rance policy," the LSTM infers that "_" is likely "u" because "insurance policy" is a common phrase, while "ins_rance" is not. The LSTM was trained on millions of text line images across hundreds of languages, learning both character shapes and language-specific patterns such as common letter sequences and word structures.

Language models further refine recognition outputs. Statistical language models, typically n-gram models trained on text corpora, score character sequences by linguistic plausibility and adjust recognition when implausible sequences are detected. For instance, if character recognition outputs "rhe" at the start of a sentence, the language model might correct it to "the," a common word, rather than accepting "rhe," which is rare or nonexistent. Since language models are language-specific, accurate language detection prior to recognition improves overall results.

Page Segmentation Modes (PSM) allow Tesseract to adapt to different document layouts. Tesseract offers multiple PSMs that control how page layout analysis proceeds. PSM 1 performs automatic page segmentation with orientation and script detection, making it suitable when document orientation or script (Latin, Arabic, Chinese) is unknown. PSM 3 applies fully automatic segmentation without orientation detection, which is the default mode and appropriate for standard documents with known orientation. PSM 6 assumes a single uniform text block, skipping block detection, and is useful for simple documents with one contiguous text region. PSM 11 finds sparse text in any order, which suits documents with scattered text, such as receipts or forms containing isolated fields. Selecting the appropriate PSM based on document type, for example full-page policy text, sparse claim forms, or multi-column layouts, significantly impacts accuracy. Incorrect PSM selection can cause missing text (failing to detect regions), garbled output (incorrect segmentation), or excessive processing time (using complex analysis for simple documents).

Confidence scores enable quality assessment and selective human verification. Tesseract provides confidence scores at multiple granularities: per-character confidence (on a 0 to 100 scale) indicates recognition certainty for each character, per-word confidence aggregates character confidences, and per-line confidence aggregates word confidences. These scores correlate closely with actual accuracy: high-confidence extractions (90% to 100%) are typically correct, while low-confidence extractions (below 60% to 70%) often contain errors.

In insurance claims processing, these confidence scores enable flagging uncertain extractions for human review, selecting between single-engine and multi-engine processing based on quality, and rejecting completely failed OCR attempts, since very low average confidence suggests the document is illegible or preprocessing has failed.

Configuration and optimization strongly affect Tesseract performance. The engine accepts numerous configuration parameters, including language specification (single or multiple languages for multilingual documents), character whitelists or blacklists (restricting recognition to expected characters, useful for fields such as policy numbers with known formats), OCR Engine Mode (legacy engine, LSTM engine, or both combined), and various image preprocessing flags. For insurance applications, typical configurations specify language (for example, Italian, English, or multilingual), use the LSTM engine mode for best accuracy, and apply character whitelists for structured fields, such as policy numbers restricted to alphanumeric characters or claim amounts restricted to digits and currency symbols. Preprocessing integration often feeds images through custom preprocessing pipelines, including binarization, deskewing, and noise reduction as discussed in chapter 2.5.3, before invoking Tesseract, thereby optimizing image quality for recognition.

### 2.5.6 Advanced Document Processing: GPT-4 Vision

While traditional OCR engines extract raw text from images, understanding document structure and extracting specific information often requires additional intelligence beyond character recognition. Multimodal Large Language Models, particularly GPT-4 Vision, represent a paradigm shift in document processing by combining visual understanding with language comprehension.

GPT-4 Vision capabilities extend beyond text recognition to document understanding. GPT-4V processes document images and can extract structured information directly, for example, “Extract all entries from this premium table as JSON,” answer content questions such as “What is the deductible mentioned in this policy page?”, understand complex layouts without explicit parsing rules by interpreting multi-column text, nested tables, or forms with non-linear reading order, and interpret visual elements such as logos indicating insurance companies, diagrams explaining coverage scenarios, or highlighting and annotations emphasizing important clauses (OpenAI, 2023). Unlike OCR engines that merely convert images to text strings, GPT-4V understands semantic content and structure, enabling higher-level information extraction.

GPT-4V offers substantial advantages over traditional OCR-plus-parsing pipelines. Context understanding allows GPT-4V to infer relationships without explicit rules. It recognizes that a number next to the word “Deductible:” represents the deductible amount, that table cells in the “Premium” column contain pricing information, and that checkboxes indicating coverage types map to specific insurance products, all without programming explicit parsing logic. Traditional pipelines require rule-based parsers, such as regular expressions to extract patterns, heuristics to locate specific fields, and layout analysis to map positions to meanings. GPT-4V replaces hundreds of lines of parsing code with natural language instructions, for example, “Extract the deductible amount from this document.”

Table extraction demonstrates GPT-4V’s strengths particularly well. Insurance policy documents often contain complex tables such as premium tables with merged cells (single headers spanning multiple columns), multi-level headers (hierarchical column groupings), conditional formatting (different pricing based on customer age, vehicle type, or coverage level), and embedded notes or footnotes. Traditional table extraction methods use rule-based techniques (detecting horizontal and vertical lines, clustering text into cells, and inferring row and column structure) or specialized models like TabNet and TableBank parsers. These approaches struggle with irregular tables containing missing borders, cells spanning rows or columns, or nested tables. GPT-4V can extract such tables by understanding visual structure and semantic content jointly. When instructed to “Extract this premium table as JSON with coverage types, customer categories, and prices,” GPT-4V produces structured output that correctly maps table contents to semantic categories, even handling merged cells and complex headers that confuse traditional parsers.

Error correction leverages GPT-4V’s language understanding. When traditional OCR produces errors, such as character misrecognitions or garbled text from poor image quality, downstream systems receive corrupted text. GPT-4V can infer correct text even when visual recognition is ambiguous by using contextual understanding. For example, if an insurance document image contains degraded text where “deductible” appears as “d_ductible” (with one letter unreadable), GPT-4V infers the missing letter from context, recognizing that “deductible” is a common word in insurance policies, while “daductible” or “diductible” are not. This type of contextual error correction is difficult for traditional OCR, which lacks semantic understanding.

Multimodal reasoning combines textual and visual information. Insurance claims often include both textual descriptions and photographs. For instance, a claim may include a written description “front bumper damage” alongside a photograph showing the actual damage. GPT-4V can analyze both, verifying consistency by asking “Does the photograph show front bumper damage as described?”, assessing damage severity from the image, and correlating damage types to policy coverage terms. Traditional pipelines process text and images separately, requiring explicit logic to relate them, whereas GPT-4V’s native multimodality enables joint reasoning.

Use cases in insurance demonstrate GPT-4V’s versatility. Extracting glossary terms from policy documents involves identifying term-definition pairs, preserving formatting such as italicized terms or bold definitions, and maintaining hierarchical structure, for example, terms within subsections. GPT-4V can extract complete glossaries while preserving this structure. Parsing premium tables with conditional pricing enables extraction of complex pricing schemes where premiums depend on multiple factors such as coverage type, customer age, vehicle value, and deductible chosen, producing structured data in formats such as JSON or CSV suitable for database insertion. Analyzing damage photographs alongside claim descriptions verifies claim consistency, assesses damage severity, and relates visual damage to coverage types mentioned in policies. Extracting structured data from varied claim form layouts also becomes simpler, as GPT-4V can handle diverse form designs used by different insurers without requiring custom parsers for each layout, generalizing across designs by understanding content semantically rather than positionally.

Limitations and cost considerations temper GPT-4V’s advantages. API costs for GPT-4V are higher than text-only models or traditional OCR, approximately $0.01 to $0.02 per image as of 2023 to 2024, compared to $0.0015 per page for Tesseract-equivalent cloud OCR or near-zero for local Tesseract. For insurance companies processing thousands of documents monthly, these costs accumulate substantially. Latency is also higher, since processing a document image with GPT-4V takes 3 to 10 seconds depending on image complexity and output requirements, compared to sub-second processing with Tesseract. Therefore, GPT-4V is typically reserved for complex documents such as tables, forms, or mixed-content pages where traditional OCR struggles, while simpler documents such as plain text pages use faster, cheaper extraction methods. Model availability and access are additional considerations, as GPT-4V requires API connectivity (introducing external dependency and potential data privacy implications for sensitive insurance documents) and is subject to rate limits and availability constraints.

For insurance claims processing, GPT-4V complements rather than replaces traditional OCR. Simple text extraction uses Tesseract or native PDF extraction, complex structured extraction involving tables or forms uses GPT-4V, and handwritten content may use specialized handwriting recognition models or cloud OCR services. This tiered approach balances cost, latency, and accuracy across diverse document types.

## 2.6 Vector Embeddings and Semantic Search

### 2.6.1 Text Embeddings: From Words to Vectors

Text embeddings, dense vector representations of textual units, enable semantic search by mapping text into continuous spaces where semantic similarity corresponds to geometric proximity. This allows documents to be found based on meaning rather than simple keyword overlap, a capability essential for insurance policy retrieval where queries and documents often use varied terminology.

Early methods such as Word2Vec [62] and GloVe [63] learned word vectors through distributional semantics, where words appearing in similar contexts receive similar embeddings. The famous example

$$\text{vec}(\text{"king"}) - \text{vec}(\text{"man"}) + \text{vec}(\text{"woman"}) \approx \text{vec}(\text{"queen"}) \tag{2.4}$$

demonstrates how these models encode semantic relationships. However, these methods assign a single vector per word regardless of context, meaning that the word “bank” has the same representation whether referring to financial institutions or river banks, which limits their applicability for ambiguous terminology frequently encountered in insurance documents.

Sentence-BERT (SBERT) [43] advanced beyond word-level representations by fine-tuning BERT models to produce semantically meaningful sentence embeddings via Siamese network training. SBERT generates 768-dimensional embeddings that can be efficiently compared using cosine similarity, enabling fast and accurate retrieval. It achieves orders-of-magnitude faster search than cross-encoder approaches while maintaining strong semantic accuracy.

Contextualized embeddings from BERT [10] further improve upon these techniques by generating different vectors for the same word in different contexts through masked language modeling. For example, “premium” in “insurance premium” receives a different embedding than “premium” in “premium quality,” thereby capturing context-dependent meanings. In insurance applications, this capability enables distinguishing domain-specific terms such as “collision coverage” (a vehicle insurance term) from the general word “collision,” significantly improving retrieval precision for specialized terminology.

### 2.6.2 OpenAI text-embedding-ada-002

This work employs OpenAI’s text-embedding-ada-002, or more recent embedding families with comparable dimensionality and cost profiles, for semantic search in insurance document retrieval [44]. The model is trained via contrastive learning on text pairs, producing 1536-dimensional embeddings in which semantically similar texts exhibit high cosine similarity.

Key characteristics include 1536 dimensions that balance semantic nuance against storage requirements (approximately 6KB per embedding, with 100,000 chunks equating to around 600MB), an 8191-token context window that comfortably accommodates typical policy chunks of 200 to 500 tokens, and L2-normalized embeddings that enable efficient dot product similarity computation. The model also demonstrates competitive performance on semantic search benchmarks such as MTEB and MS MARCO, while its API availability eliminates the need for local infrastructure.

For insurance applications, the model effectively handles synonym matching, for instance “glass damage” retrieving related terms such as “windshield,” “crystal,” or “lunotto,” captures domain-specific terminology relationships like “deductible” and “franchigia” or “premium” and “premio,” and supports limited cross-lingual retrieval. The cost is approximately $0.0001 per 1000 tokens (as of 2023), meaning that embedding a 100,000-chunk corpus costs around $10 to $20, which is economical for stable policy corpora requiring infrequent re-embedding.

However, several limitations remain. API dependency introduces latency, typically around 50 to 200 milliseconds of network overhead, and raises data privacy considerations since documents are transmitted to external servers. The model also cannot be fine-tuned on domain-specific data, and its internal workings are opaque, hindering detailed error analysis. Organizations with substantial proprietary data and existing GPU infrastructure may achieve superior results with fine-tuned open-source alternatives, although API-based approaches continue to offer significant operational simplicity for moderate-scale deployments.

### 2.6.3 ChromaDB: Vector Database for LLM Applications

ChromaDB is an open-source vector database designed specifically for LLM applications, prioritizing developer experience and rapid prototyping while maintaining production viability. Unlike general-purpose systems such as Pinecone, Weaviate, or Milvus, ChromaDB focuses on Retrieval-Augmented Generation (RAG) workflows, offering a level of simplicity particularly suitable for insurance claims processing applications.

Key features include a straightforward Python API that abstracts vector search complexities, allowing creation of collections, document insertion, and querying with minimal code. Its in-process execution enables local development without the need for separate database servers, while native LangChain integration reduces boilerplate code. Metadata filtering supports SQL-like where clauses, combining semantic search with structured predicates such as customer_id or policy_type. Collections act as logical namespaces, supporting a dual vector store architecture that distinguishes between "permanent_policies" for contract documents and "temp_uploads" for session-scoped claim documents. Persistence options range from in-memory mode, which is fastest but ephemeral, to local file-based mode using SQLite and HNSW index files, and to client-server mode for production deployments.

ChromaDB employs HNSW (Hierarchical Navigable Small World) indexing [46] for approximate nearest neighbor search. HNSW organizes vectors within multi-layer graphs where upper layers facilitate long-range navigation and lower layers refine searches, achieving $O(\log N)$ complexity. Querying one million 1536-dimensional vectors typically takes between 10 and 50 milliseconds on modern hardware, depending on parameter configurations. Tunable parameters M (edges per vertex) and ef (beam width) balance index size, construction time, search quality, and latency. For insurance applications, where recall is more important than microsecond-level latency differences, ChromaDB's quality-prioritizing defaults are well suited, with typical retrieval latencies ranging from 20 to 100 milliseconds.

### 2.6.4 Dual Vectorstore Architecture

Insurance claims processing handles two document categories with fundamentally different characteristics. Policy documents, such as contracts, coverage terms, and regulatory texts, are long-lived, shared across users, and updated infrequently. Customer-uploaded documents, including claim forms, damage photographs, and receipts, are ephemeral, user-specific, and relevant only during claim processing. A dual vector store architecture, which maintains separate vector databases for these two categories, offers substantial advantages.

The permanent vector store persistently stores policy document embeddings. Once embedded, these documents remain valid until policy updates occur, typically on an annual or quarterly basis, enabling reuse across thousands of claims without the need for re-embedding. All users query the same shared knowledge base, for example 50,000 chunks derived from 100 insurance products multiplied by 50 sections and 10 chunks each. Incremental updates, such as adding, modifying, or removing chunks, minimize downtime when policies change. Built during system initialization and persisted to disk, the permanent vector store functions as stable, long-term infrastructure.

The temporary vector store, in contrast, stores customer-upload embeddings ephemerally. Created on demand when documents are uploaded and deleted after claim processing or session timeout, these session-specific collections provide user isolation, ensuring that Customer A's documents are never retrievable by Customer B, and accommodate variable sizes ranging from 10 to 50 chunks for simple claims to several hundred for complex ones. This ephemeral design also satisfies GDPR's right to erasure by automatically deleting personally identifiable information once the claim is resolved.

Architectural separation provides multiple benefits. Data privacy and compliance improve because uploads containing personal information are retained only as long as necessary, while generic policy texts persist indefinitely. Performance remains predictable since the permanent vector store maintains nearly constant size, given that policies evolve slowly, while temporary vector stores stay relatively small, containing hundreds to thousands of chunks, thereby avoiding unbounded index growth that could degrade query latency. Operational reliability also increases, as maintenance operations on the permanent vector store, such as policy updates or re-embedding, pose no risk to customer uploads, and potential bugs in temporary document processing cannot affect the shared knowledge base. Query flexibility is enhanced as well, allowing different retrieval strategies to be applied: pure semantic similarity for policy documents versus hybrid approaches, combining semantic similarity with metadata filtering by session, for user uploads. Selective querying enables routing policy-related questions to the permanent vector store, claim-specific questions to temporary stores, or querying both with result merging when appropriate.

Implementation requires query routing logic to determine which vector store or combination of stores to query, cleanup policies to prevent resource exhaustion from concurrent temporary vector stores, and session management to handle reopened claims or multi-session workflows. Despite these additional complexities, the dual vector store architecture aligns naturally with insurance claims processing, where the distinction between shared policy knowledge and ephemeral claim documents is clear and the benefits of separation are substantial.

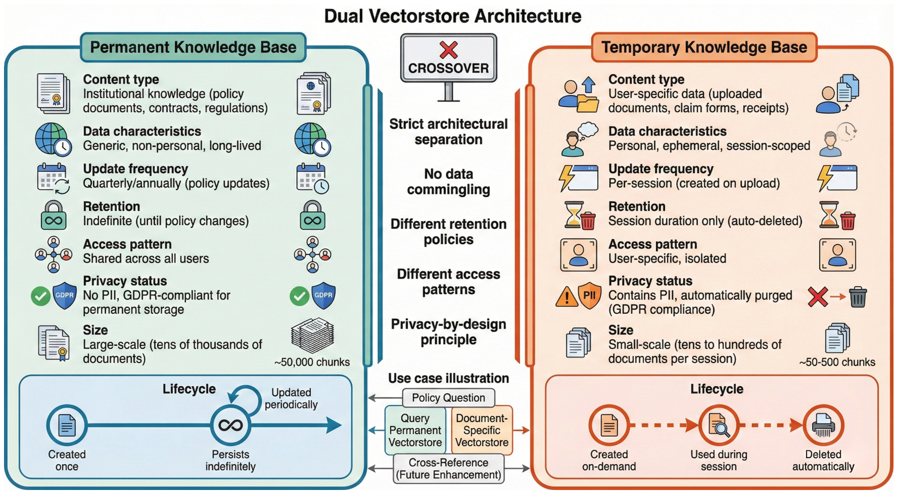
Dual Vectorstore Architecture
Permanent Knowledge Base
Content type
Institutional knowledge (policy documents, contracts, regulations)
Data characteristics
Generic, non-personal, long-lived
Update frequency
Quarterly/annually (policy updates)
Retention
Indefinite (until policy changes)
Access pattern
Shared across all users
Privacy status
No PII, GDPR-compliant for permanent storage
GDPR
Size
Large-scale (tens of thousands of documents)
~50,000 chunks
Lifecycle
Updated periodically
Created once
Persists indefinitely
CROSSOVER
Strict architectural separation
No data commingling
Different retention policies
Different access patterns
Privacy-by-design principle
Use case illustration
Policy Question
Query Permanent Vectorstore
Document-Specific Vectorstore
Cross-Reference (Future Enhancement)
Temporary Knowledge Base
Content type
User-specific data (uploaded documents, claim forms, receipts)
Data characteristics
Personal, ephemeral, session-scoped
Update frequency
Per-session (created on upload)
Retention
Session duration only (auto-deleted)
Access pattern
User-specific, isolated
Privacy status
Contains PII, automatically purged (GDPR compliance)
PII
Size
Small-scale (tens to hundreds of documents per session)
~50-500 chunks
Lifecycle
Created on-demand
Used during session
Deleted automatically

**Figure 2.2:** Theoretical separation between permanent knowledge base (policy documents) and temporary knowledge base (session-scoped customer uploads), showing lifecycle, retention policies, and privacy compliance characteristics

### 2.6.5 Chunking Strategies for Insurance Documents

Document chunking, the process of splitting long documents into segments for embedding and retrieval, critically impacts Retrieval-Augmented Generation (RAG) performance. Insurance policy documents, which often span tens of thousands of words, must be segmented into semantically coherent units that balance retrieval precision with context completeness.

Chunk size involves fundamental trade-offs. Small chunks of around 100 to 200 tokens provide precise retrieval of specific facts but offer limited context and may fragment coherent explanations across multiple chunks. In contrast, large chunks of 500 to 1000 tokens provide richer context for reasoning but reduce embedding specificity and quickly consume the context windows of large language models. Empirical findings suggest that a range of 256 to 512 tokens balances these concerns effectively for question-answering tasks. For insurance policies, where clauses typically span one to three paragraphs, chunk sizes between 300 and 500 tokens with overlap capture complete clauses while maintaining retrieval precision.

Semantic chunking respects document structure by splitting at natural boundaries such as section headers, paragraph breaks, or clause separators, rather than at fixed character counts that risk fragmenting meaning. For hierarchically structured insurance policies, hybrid approaches work best. These approaches split at subsection boundaries, for example "Glass Coverage" or "Deductible Provisions," but further divide subsections exceeding threshold lengths, such as 800 tokens, at paragraph boundaries. This ensures that chunks remain appropriately sized without breaking coherent content.

Chunk overlap, typically 10 to 20 percent (or about 50 to 100 tokens for a 500-token chunk), prevents the loss of information that spans chunk boundaries. Without overlap, related content such as “Glass coverage applies to windshield, side windows, and rear window. Deductible is €100 per incident” might be split mid-sentence, severing the conceptual relationship between coverage and deductible. Although overlap increases storage and retrieval costs, it usually provides net benefits by preventing boundary-related information loss.

Metadata enrichment adds valuable context beyond the chunk content itself. Hierarchical structure annotations, including section and subsection identifiers, document-level metadata such as product type, effective date, and jurisdiction, and chunk-specific metadata such as content type or keywords, all enhance filtering, provide richer generation context, and improve ranking accuracy. ChromaDB’s native metadata support simplifies this process through key-value dictionaries and SQL-like filtering.

Insurance-specific considerations further refine chunking strategies. Tables, including premium tables and coverage limits, require special handling, which may involve treating them as single chunks, using specialized parsers, or processing them with multimodal models. Cross-references between clauses benefit from metadata links that maintain coherence across related chunks. Glossaries should be chunked to ensure that each term-definition pair remains complete within a single chunk, preserving semantic integrity. Multilingual policies also require consistent chunking across languages, with language metadata included to enable accurate cross-lingual queries.

## 2.7 LangChain Framework

### 2.7.1 LangChain Overview

LangChain is an open-source framework simplifying LLM application development [64]. Released in late 2022, it provides high-level abstractions for recurring patterns: constructing prompts with dynamic content, chaining LLM calls, retrieving context from external knowledge bases, maintaining conversation state, and handling errors. These abstractions enable focusing on application logic rather than low-level API interactions.

LangChain’s design philosophy emphasizes composability (building complex workflows from simple components), modularity (loosely coupled interfaces enabling component swapping), and abstraction (unified interfaces for diverse backends like ChromaDB, Pinecone, Weaviate, FAISS). Components support rapid prototyping (in-memory stores, synchronous execution) and production deployment (persistent storage, async execution, monitoring callbacks).

Core abstractions include Models (interfaces to LLMs and embedding models supporting OpenAI, Anthropic, Cohere, open-source models), Prompts (tools for managing templates with variable substitution, few-shot examples, chat message sequencing), Chains (sequences of operations from simple LLM calls to complex workflows like ‘RetrievalQA‘ and ‘ConversationalRetrievalChain‘), Memory (systems maintaining conversation state via buffer memory, summary memory, entity memory), and Retrievers (interfaces abstracting over vector databases, search engines, SQL databases, supporting re-ranking, filtering, and fusion).

### 2.7.2 RAG Implementation with LangChain

LangChain provides specialized components that handle each stage of the Retrieval-Augmented Generation (RAG) pipeline, substantially reducing engineering effort compared to raw LLM API implementations.

Document loaders abstract format-specific parsing for more than one hundred formats. In insurance applications, PDF loaders such as PyPDFLoader, PDFMinerLoader, and UnstructuredPDFLoader process policies with varying layouts, CSV loaders handle customer records and claims history, and database loaders integrate directly with SQL systems including PostgreSQL, MySQL, and SQLite. All loaders return standardized Document objects containing page_content and metadata, which enables uniform processing regardless of the original source format.

Text splitters partition documents into manageable chunks for embedding. The RecursiveCharacterTextSplitter attempts different separators, such as "", "", and " ", in order, preserving semantic coherence by preferring paragraph boundaries over sentence or word boundaries. The TokenTextSplitter instead splits text by token count using tiktoken, ensuring that chunks respect model context limits. For insurance policy documents, using the RecursiveCharacterTextSplitter with token-based measurement and an overlap of 10 to 20 percent preserves clause boundaries while maintaining semantic completeness.

Vector store integrations provide standardized VectorStore interfaces supporting local options such as ChromaDB and FAISS, cloud services like Pinecone and Weaviate, and existing enterprise infrastructure including Elasticsearch or PostgreSQL with pgvector. This abstraction allows rapid prototyping with ChromaDB in local environments and seamless deployment to managed services in production with minimal code modification.

Retrieval chains encapsulate complete RAG workflows. The RetrievalQA chain orchestrates the process of querying vector stores, combining the query and retrieved documents into prompts, calling the LLM, and returning answers with optional source citations. The ConversationalRetrievalChain extends this capability by adding conversation memory for multi-turn dialogues in which follow-up questions reference prior context, an essential feature for insurance customer interactions where dialogue continuity is critical.

### 2.7.3 Prompt Engineering with LangChain

Prompt engineering, the practice of carefully designing prompts to elicit desired LLM behavior, is crucial for ensuring application quality. LangChain systematizes this process through tools that enable maintainable, versioned, and testable prompts.

The PromptTemplate component separates prompt structure from runtime data through variable placeholders that are filled at execution time. This design allows centralized prompt management, version control, and template testing using mock variables, ensuring consistent LLM behavior across queries. It also includes validation to detect missing variables before the LLM is invoked. The ChatPromptTemplate extends this capability by structuring prompts as ordered message sequences, system, user, and assistant messages, thereby aligning with the chat-optimized training paradigms of models such as GPT-3.5-turbo and GPT-4.

The FewShotPromptTemplate manages examples demonstrating desired input-output behavior, guiding the LLM toward correct response patterns. In insurance applications that require precise output formats, such as JSON with defined fields, few-shot examples can illustrate field extraction, numerical formatting for currency and percentages, expression of uncertainty, and citation of relevant policy clauses. The template dynamically selects which examples to include when the full set exceeds the prompt's capacity, ensuring adaptability to context constraints.

Output parsers handle the structured extraction of data from model responses. The PydanticOutputParser integrates with Pydantic to perform schema-based validation, while the JSONOutputParser extracts valid JSON from free-form LLM outputs, handling markdown fences and formatting inconsistencies. Specialized parsers also exist for lists, datetimes, and enumerations. In insurance claims processing, robust parsing ensures that extracted values such as amounts, dates, and customer IDs are correctly formatted and validated before database insertion, with retry or escalation logic triggered when parsing fails.

### 2.7.4 Conversation Memory and Context Management

Interactive insurance claims processing requires multi-turn dialogues; however, large language models (LLMs) are stateless and lack inherent memory of previous interactions.

Conversation memory systems address this limitation by maintaining state across dialogue turns, thereby enabling natural and coherent exchanges. Without such memory, follow-up questions like “What about for comprehensive coverage?”

posed after an initial query such as “What is my glass coverage deductible?” would lack necessary context and could not be accurately answered.

LangChain provides several memory implementations, each with distinct trade-offs. Conversation buffer memory stores all messages and injects the complete dialogue history into subsequent prompts, ensuring full context retention but consuming tokens linearly with conversation length. For example, a ten-turn conversation may require 2000 to 5000 tokens solely for history, eventually exceeding model context limits. This approach suits short conversations or high-stakes applications that demand complete context. Conversation summary memory, by contrast, periodically summarizes older exchanges using the LLM, maintaining bounded token consumption that supports indefinite conversations, though at the cost of potential information loss and added summarization latency and expense. Entity memory extracts and tracks specific entities such as customer IDs, policy numbers, and dates rather than retaining raw conversation text. This method is highly efficient in token usage but lossy for information that falls outside defined entity categories. Windowed memory retains only a fixed number of recent exchanges, for instance the last five turns, providing predictable and bounded token use sufficient for maintaining immediate context, while sacrificing access to longer-range dependencies.

For insurance claims processing, hybrid approaches tend to perform best. Entity memory can be used to persistently track critical identifiers such as customer IDs and policy numbers, windowed memory can preserve conversational flow across the most recent three to five turns, and manual state management within application logic can handle workflow tracking, ensuring that conversation stages are controlled explicitly rather than inferred by the LLM. This combination balances accuracy, since critical facts are never lost, with efficiency, through bounded token usage, and usability, by maintaining a natural conversational experience.

### 2.7.5 LangChain for Production Deployments

Production deployment requires addressing scalability, reliability, observability, and cost management. LangChain provides features that support these requirements, although achieving production readiness still demands careful configuration beyond default behaviors.

Asynchronous support enables efficient concurrent request handling. Insurance claims systems must serve multiple users simultaneously, and synchronous code, where each request blocks until completion, does not scale effectively. A slow LLM call taking 2 to 5 seconds would block server threads and significantly limit throughput. LangChain’s asynchronous variants, such as chain.ainvoke and vectorstore.asimilarity_search, leverage Python’s asyncio framework to enable concurrent execution, allowing a single server process to handle dozens or even hundreds of concurrent queries. For FastAPI deployments, using asynchronous methods consistently throughout the request pipeline is essential to maintain performance under load.

Callbacks provide observability hooks into execution stages. Custom callback functions can log operations for debugging and audit trails, which are critical for insurance compliance, measure performance by tracking latency at each stage, including retrieval and LLM inference, monitor costs by aggregating token usage for budget control, and detect anomalies such as zero retrieved documents, timeouts, or unexpected response lengths. For insurance applications processing thousands of claims monthly, active cost monitoring prevents uncontrolled API usage that could otherwise result in significant expenses.

Caching mechanisms optimize both cost and latency for repetitive queries that are common in insurance contexts, for instance, when multiple customers ask "What is the deductible for glass coverage?" for the same policy type. Exact match caching returns stored responses for identical prompts, semantic caching addresses query variations by leveraging embedding similarity, and intermediate result caching stores retrieved documents for frequently asked questions. For insurance applications, hybrid caching strategies, caching general policy information while excluding customer-specific data, offer an effective balance between cost savings and response correctness.

Error handling mechanisms ensure system robustness. These include retry logic with exponential backoff for transient failures, fallback models that substitute GPT-3.5-turbo when GPT-4 is unavailable, circuit breakers that prevent cascading failures, and user-facing error messages that translate technical errors into clear, actionable feedback.

Security and data privacy are paramount for handling sensitive insurance data. Best practices include enforcing HTTPS for encrypted data transmission, encrypting storage at rest, implementing strict access controls such as customer-level query filtering, managing data retention through automatic deletion of customer uploads in compliance with GDPR and CCPA (via temporary vector store deletion), redacting personally identifiable information from logs, and securely managing API keys. While LangChain simplifies many of these concerns through built-in abstractions, deploying robust insurance systems ultimately requires a thoughtfully designed architecture that extends beyond the framework itself.

## 2.8 Human-in-the-Loop AI Systems

### 2.8.1 Motivation for Human Oversight

Fully autonomous claims processing faces significant challenges. Insurance claims involve substantial financial decisions, and errors can lead to customer dissatisfaction, financial losses, or legal liabilities. Human oversight is therefore essential to ensure that critical decisions are validated before finalization. Moreover, the insurance industry is heavily regulated, and many jurisdictions explicitly require human participation in claim decisions, particularly in cases involving denials or high-value settlements. AI systems must therefore produce auditable decision trails suitable for human review, providing clear explanations of how each conclusion was reached.

AI systems also struggle with unusual situations, ambiguous policy language, or novel scenarios that fall outside their training data, whereas human adjusters excel at interpreting exceptional cases and exercising judgment. Customers are generally more accepting of AI-assisted outcomes when they know that a human expert has reviewed and approved the final decision.

Under regulatory frameworks such as the GDPR's right to explanation and the EU Insurance Distribution Directive (IDD), automated decisions that affect customers must be both explainable and subject to human review. Retrieval-Augmented Generation (RAG) architectures address these requirements effectively through explicit source attribution and human-in-the-loop oversight. Each claim decision can be traced to specific policy clauses via retrieved documents, forming a transparent audit trail that human adjusters can verify. This approach balances automation efficiency with regulatory obligations for transparency, accountability, and human governance.

### 2.8.2 Levels of Automation

Human-in-the-loop systems can be designed with varying levels of automation. At Level 1 (AI-Assisted), the AI provides information and recommendations while humans make all final decisions. For example, optical character recognition (OCR) may extract claim data, but adjusters manually verify and process the information. Level 2 (AI-Recommended) allows the AI to propose a decision accompanied by confidence scores, which humans then review and either approve or override. An illustrative case is when the AI calculates a reimbursement amount and the adjuster examines the calculation steps before granting approval. Level 3 (Conditional Autonomy) enables the AI to process claims autonomously for simple, high-confidence cases while escalating complex or uncertain ones to human experts. For instance, claims under €500 with no detected anomalies might be automatically approved, whereas others would require human review. Finally, Level 4 (Full Autonomy with Audit) represents a fully automated process in which the AI handles all claims independently, while humans conduct periodic audits on sample cases to ensure quality control and facilitate continuous improvement.

For the insurance domain, automation levels corresponding to Level 2 or Level 3 typically provide the optimal balance between operational efficiency and effective risk management.

### 2.8.3 Design Patterns for Human-AI Collaboration

Designing effective collaboration between human experts and AI systems requires careful consideration of interaction patterns, information flow, and the calibration of trust. Insights from human–computer interaction research and practical deployments in high-stakes domains such as insurance, healthcare, and legal services have given rise to several recurring design patterns that guide the development of such systems.

A central principle in these designs is confidence-based escalation, in which AI systems quantify their uncertainty and use these estimates to determine when human oversight should intervene. In the context of insurance claims processing, confidence scoring can incorporate multiple elements. Retrieval confidence assesses how semantically similar the retrieved policy clauses are to the query: high similarity indicates that the retrieved information is likely relevant, while low similarity suggests that the question may concern a policy gap or edge case that warrants human judgment. Extraction confidence, relevant to OCR-based document processing, evaluates character- and field-level reliability of data such as policy numbers, claim amounts, and dates, identifying cases that may require human verification. Another key factor is answer consistency, which measures whether the system provides stable responses when queried multiple times with slightly different prompts; significant variance reveals uncertainty that should prompt review. Finally, policy completeness examines whether the policy explicitly addresses the claim scenario or whether interpretation of ambiguous or unstated clauses is required, again indicating the need for escalation to a human expert.

Determining appropriate confidence thresholds is critical. Excessively sensitive thresholds can overwhelm human reviewers by escalating too many cases, thereby reducing automation benefits, whereas overly permissive thresholds risk allowing uncertain or erroneous cases to be processed autonomously. The optimal balance depends on the relative costs of different types of errors (such as false approvals versus false denials), available human capacity, and acceptable levels of operational risk. In practice, thresholds are best refined iteratively through human feedback: cases in which human reviewers disagree with supposedly high-confidence AI decisions provide valuable data for calibration and continuous improvement.

Another key dimension of human–AI collaboration lies in explainable AI and transparent reasoning. For human experts to effectively review and validate AI outputs, they must be able to understand the reasoning process behind each decision. Retrieval-Augmented Generation (RAG) systems inherently offer advantages in this regard because they ground responses in explicit, retrievable sources. Each claim decision can thus cite the specific policy clauses from which its reasoning derives, allowing human reviewers to verify that the citations are relevant, correctly interpreted, and appropriately applied. When performing reimbursement calculations, for instance, the system should expose its reasoning in an interpretable sequence of operations: “Claim amount €950 $\rightarrow$ Apply coverage limit (min(€950, €1000) = €950) $\rightarrow$ Subtract deductible (€950 – €200 = €750) $\rightarrow$ Apply co-pay (€750 $\times$ 0% = €0) $\rightarrow$ Final reimbursement €750.” Each computational step in this trace should correspond to a verifiable policy clause. Moreover, the system should decompose its overall confidence into components such as retrieval quality, OCR accuracy, and policy applicability, enabling human reviewers to identify weak points in the reasoning chain. When policy language is ambiguous, the system may also present alternative interpretations, for example, noting that a case could be treated as vandalism (covered under Clause 3.2) or intentional damage (excluded under Clause 5.1), and recommend human review to determine intent.

Explainability must be balanced between completeness and conciseness. Presenting excessive detail, such as full LLM prompts, every retrieved document, or exhaustive calculation traces, can overwhelm human reviewers and obscure the essential reasoning. Progressive disclosure, where a concise summary is presented first with options to explore deeper layers of evidence, generally achieves better usability and cognitive efficiency.

A further principle involves interactive refinement and learning from human feedback. Human oversight not only serves as a safeguard but also provides a rich source of information for improving AI performance. Effective systems capture and learn from both explicit and implicit feedback. Explicit feedback arises when human reviewers override AI decisions, often with short justifications such as “Denied because damage predates policy start date (not detected by AI),” which helps identify specific failure modes. Implicit feedback can be inferred from human behavior: cases that require extended review time suggest model uncertainty, while rapid approvals imply reliable AI decisions. Aggregated across many users, these signals can reveal systematic weaknesses, for example, frequent overrides for certain coverage types or document formats may point to the need for more targeted retrieval or improved OCR preprocessing. In large organizations, this information can further support Reinforcement Learning from Human Feedback (RLHF), where human corrections are used to train reward models that fine-tune system behavior. Although RLHF requires substantial amounts of labeled data and is practical primarily for enterprises processing high claim volumes, it remains a powerful strategy for iterative system enhancement.

Equally important is workflow integration and cognitive fit. Effective AI systems should complement, rather than replace, existing human workflows. In insurance claims processing, for example, the workflow typically proceeds through stages of intake, document review, coverage verification, valuation, and approval or denial. AI should augment each stage by performing tasks such as automatic data extraction during intake, highlighting relevant policy clauses during review, and calculating valuations during assessment, all while preserving human control over final decisions. Outputs must also be tailored to the information needs of specific roles, since claims adjusters, underwriters, fraud investigators, and customer service representatives require different types of contextual information. Moreover, systems must respect professional expertise: experienced adjusters rely on years of domain-specific heuristics and intuition, and AI systems that override such expertise without transparent justification risk eroding trust. Framing AI as a "decision support" or "second opinion" tool, rather than as an authoritative decision-maker, aligns better with professional culture and promotes adoption. Positioning AI as a supportive technology that allows experts to handle a larger volume of cases, focus on complex scenarios, and reduce time spent on repetitive tasks such as data entry and document retrieval further strengthens trust and user acceptance.

Finally, alert systems and anomaly detection play a vital role in ensuring oversight and risk management. Beyond routine case processing, AI systems should identify potential anomalies that require human investigation. These may include policy issues such as expired policies, coverage gaps, or exclusions matching claim descriptions (for instance, "Policy excludes water damage; claim mentions flooding"), data inconsistencies such as mismatches between claimed amounts and documentation, or duplicate claims for the same incident. Fraud-related patterns may also trigger alerts, such as claims filed immediately after policy purchase, exaggerated damage descriptions inconsistent with photographs, or histories of frequent claims. High-value cases exceeding predefined thresholds, such as €5000, should automatically be routed to senior adjusters regardless of AI confidence.

Alert systems must be carefully tuned to balance sensitivity and specificity. Excessive false positives can desensitize human reviewers, diminishing the system's credibility, while false negatives allow problematic cases to pass unnoticed. Achieving optimal performance requires ongoing monitoring, regular threshold adjustment, and feedback loops linking operational outcomes to alert calibration.

Through these combined mechanisms, confidence-based escalation, transparent reasoning, feedback-driven refinement, workflow alignment, and proactive anomaly detection, human-in-the-loop insurance systems can achieve an effective balance between automation efficiency, expert oversight, and regulatory compliance.

### 2.8.4 Human-AI Interface Design

The design of interfaces mediating human–AI interaction profoundly influences system effectiveness, user satisfaction, and error rates. Poorly designed interfaces can undermine even the most sophisticated AI models by making verification cumbersome, obscuring reasoning processes, or failing to accommodate expert workflows. Research in human–computer interaction and explainable AI [65] [66] provides established principles for creating interfaces that support effective, transparent, and trustworthy decision making in AI-assisted systems.

In insurance claims processing, adjusters must review numerous cases each day, making information architecture and dashboard design essential to efficiency and accuracy. Interfaces should present essential information immediately while also allowing deeper investigation when necessary. The primary information layer must display all critical decision-relevant content, such as claim status, AI recommendations, extracted data, validation results, and risk indicators, without requiring scrolling or complex navigation. Clear visual cues, including consistent color semantics (red for problems, yellow for caution, green for normal conditions), structured layouts, and intuitive grouping of related elements, help adjusters identify the claim's current state and the AI's recommended action at a glance. When well designed, dashboards allow experienced professionals to make routine decisions within seconds while still signaling when a case demands closer review.

Beyond the summary layer, interfaces should enable detailed examination through progressive disclosure, revealing additional information only when requested. This design principle allows users to verify AI outputs without being overwhelmed by unnecessary data. For example, when a claim decision is selected, retrieved documents and their similarity scores should appear alongside highlighted excerpts of relevant text so that adjusters can confirm retrieval accuracy and relevance. OCR results should be shown with the original document images and confidence scores for each field, highlighting uncertain extractions and allowing users to correct errors directly. When the AI provides reimbursement recommendations, the interface should make the reasoning trace explicit, showing the entire calculation process, including coverage limits, deductibles, co-pay percentages, and intermediate values, each linked to the policy clauses defining them. Ambiguous cases benefit from the display of alternative interpretations, for instance, distinguishing between vandalism (covered) and intentional damage (excluded), with a prompt for human judgment. Advanced users should also have the option to access complete contextual information such as full policy documents, raw API responses, and system logs, which supports transparency, training, and dispute resolution. Progressive disclosure thus prevents information overload, allowing novices to focus on essential elements while giving experts the means to drill into fine-grained details when necessary.

Human override mechanisms form another critical aspect of interface design. Adjusters must be able to correct AI recommendations easily, yet every override should be traceable and justified. Interfaces should provide intuitive controls for modifying outcomes, for example, changing an approval to a denial or adjusting reimbursement amounts, without introducing excessive friction that might discourage necessary corrections. When an override occurs, the system should request a concise justification, supported by predefined categories and optional free-text input. This process not only fosters accountability but also generates structured feedback that can be analyzed to identify recurring AI failure modes.

Comprehensive audit trails recording the timestamp, user identity, and rationale for each override support both regulatory compliance, by demonstrating human oversight, and organizational learning, by highlighting systematic discrepancies between human and AI reasoning.

Managing cognitive load is equally important, as insurance claims involve numerous complex and interdependent data points, including policy terms, coverage limits, deductibles, exclusions, claim details, document content, and customer history. Presenting all of this information simultaneously would exceed human cognitive capacity and degrade performance.

Effective interfaces therefore emphasize focus through visual salience, using text weight, color, and position to draw attention to the most relevant elements for the decision at hand, and minimize clutter by showing only decision-critical data by default while keeping technical metadata or peripheral details hidden until needed. They also respect the limitations of working memory, which research suggests can hold approximately four to seven items at once [67], by structuring dashboards so that no more than a handful of key decision factors are visible on the main view. Additionally, error-prevention mechanisms such as form validation, confirmation prompts for irreversible actions, and sanity checks ensure that users do not inadvertently approve claims exceeding coverage limits or enter invalid figures.

Research on decision support systems [68] [69] consistently shows that well-designed interfaces enhance decision accuracy, reduce processing time, and increase user satisfaction. These outcomes are particularly critical in insurance claims processing, where both efficiency and accuracy directly affect customer experience, compliance, and financial outcomes. A carefully designed interface therefore acts not merely as a display mechanism but as a central component of human–AI collaboration, shaping the reliability and accountability of the entire decision-making process.

### 2.8.5 Evaluation Considerations

Human-in-the-loop systems must be evaluated across several interrelated dimensions. The accuracy of optical character recognition directly influences the quality of extracted data, while the faithfulness of Retrieval-Augmented Generation determines the reliability of the system's answers. Overall system performance affects operational efficiency, and the degree of human–AI agreement provides a measure of the system's practical utility in real-world workflows. These evaluation aspects are examined in detail in the empirical analysis presented in Chapter 4.

## 2.9 Technology Selection Criteria

This section outlines the criteria and rationale that guided the selection of technologies for AI-powered claims processing. Specific implementation details and the description of architectural integration are presented in Chapter 3.

### 2.9.1 Model Selection Criteria

Model selection for insurance claims processing requires a careful balance among reasoning complexity, latency tolerance, cost efficiency, multimodal capabilities, and context window requirements. Effective systems rarely rely on a single model; instead, they employ multiple models, each assigned to tasks that align with its particular strengths.

Tasks that involve complex reasoning, such as policy interpretation with conditional logic, multi-step reimbursement calculations, or detection of contradictions across multiple documents, benefit from the superior analytical depth and long-context consistency of GPT-4, which consistently outperforms other models on multi-step reasoning benchmarks. Simpler extraction tasks, including identifying customer IDs, extracting dates, or classifying claim types, can be handled effectively by GPT-3.5-turbo with negligible differences in quality. Tiered configurations that delegate extraction and classification to GPT-3.5-turbo while reserving GPT-4 for reasoning and interpretation strike a practical balance between performance, cost, and latency.

Latency tolerance varies across stakeholders and use cases. Customer-facing chatbots, for instance, require rapid response times, as delays exceeding five to seven seconds are typically perceived as sluggish. Such applications therefore favor GPT-3.5-turbo, which achieves latencies of one to two seconds, over GPT-4, which typically responds within three to five seconds. In contrast, claims adjusters reviewing cases can tolerate higher latencies, often between ten and twenty seconds, in exchange for greater accuracy, as they typically work on multiple claims concurrently. Batch-processing workflows, where throughput is more important than individual response time, can leverage slower but more thorough models or even multi-engine ensembles to maximize overall efficiency.

Cost considerations remain a decisive factor in large-scale insurance operations. GPT-4's cost ranges from \$0.03 to \$0.06 per 1,000 tokens, compared to \$0.001

to \$0.002 for GPT-3.5-turbo. A typical insurance retrieval-augmented generation (RAG) query, encompassing 2,000 to 5,000 tokens of retrieved policy context, customer data, and model-generated response, therefore costs between \$0.10 and \$0.25 when processed with GPT-4, but only \$0.01 to \$0.05 with GPT-3.5-turbo. At a volume of 10,000 monthly queries, this difference translates to operational costs of approximately \$1,000 to \$2,500 versus \$10 to \$50. To manage such disparities, systems can employ several optimization strategies, including selective model routing (reserving GPT-4 for high-value or high-risk cases), prompt optimization (reducing token usage from 3,000 to 2,000 yields roughly a 33% cost reduction), caching frequent queries (potentially reducing LLM calls by 30–70%), and hybridization with lightweight tools such as rule-based regular expressions for policy number extraction or small classifiers for document-type identification, reserving LLMs for tasks requiring language understanding and reasoning.

Multimodal requirements further dictate model routing strategies. Text-only tasks, such as policy question answering, claim summarization, or structured data extraction from textual sources, are efficiently handled by text-only models, which are both faster and less expensive. However, tasks that require visual reasoning, including the analysis of damage photographs, extraction from complex document layouts, or the interpretation of handwriting, necessitate the use of GPT-4 Vision. Selectively invoking multimodal models in this way minimizes expensive API usage while ensuring that visual inputs are accurately processed.

Finally, context window requirements depend on the complexity of the input documents. Moderate context capacities, such as the 8k to 16k token limits available in GPT-3.5-turbo and standard GPT-4, are sufficient for most insurance queries. These typically involve the retrieval of three to five policy chunks of approximately 400 to 500 tokens each, combined with customer data and task instructions, all of which fit comfortably within the 8k-token limit. Larger context windows of 32k tokens or more enable reasoning across full policies or comprehensive document sets but come at the expense of increased inference time and cost. In edge cases requiring extensive context, it is often more efficient to employ iterative retrieval and summarization or to escalate the task for human review rather than relying solely on large-context models.

### 2.9.2 RAG vs. Fine-Tuning: Strategic Considerations

Adapting general-purpose large language models (LLMs) to the insurance domain requires selecting between two principal strategies: Retrieval-Augmented Generation (RAG), which provides domain knowledge through dynamically retrieved context, and fine-tuning, which embeds domain knowledge directly into model parameters. Each approach entails distinct trade-offs that influence system architecture, maintenance requirements, and overall performance.

RAG-based systems retain the base model's parameters in their original state while augmenting each prompt with relevant information retrieved from external sources. This design offers a major advantage for domains such as insurance, where policies, coverage terms, and regulatory frameworks evolve frequently, often on annual or quarterly cycles. Because RAG relies on external retrieval, updates to policy content are immediately reflected in system outputs simply by refreshing the vector database, eliminating the need for model retraining. In contrast, fine-tuned systems require the collection and curation of new training data, followed by retraining that can take hours or even days, and subsequent redeployment, an operationally burdensome process for environments subject to frequent change.

The cost differential between these approaches is also significant. RAG primarily incurs a one-time cost for document embedding, typically around ten to twenty dollars for a 100,000-chunk corpus, while fine-tuning requires thousands of carefully labeled examples, substantial GPU computation, and additional engineering effort. Moreover, RAG offers inherent advantages in transparency and regulatory compliance: because retrieved text is explicitly cited, users can trace each output back to its source, for example, a statement such as "Your glass deductible is €100" can be directly linked to the specific policy clause from which it was drawn. Fine-tuned models, by contrast, internalize such information within their parameters, making it effectively opaque and difficult to verify, an important limitation in regulated industries requiring auditability and explainability.

RAG also accommodates heterogeneous information sources, such as policies, legal regulations, procedural guidelines, and frequently asked questions, all of which can coexist within a single retrieval framework. Fine-tuning, on the other hand, requires all such knowledge to be represented as training examples, complicating data preparation and limiting adaptability. Nonetheless, RAG's performance is bounded by the quality of its retrieval mechanism; if retrieval fails to identify the most relevant content, model accuracy suffers.

Furthermore, the retrieval process introduces additional latency, typically between twenty and one hundred milliseconds, although this overhead is often negligible relative to overall inference time.

Fine-tuning offers complementary advantages by embedding domain knowledge directly into model parameters. This eliminates retrieval latency, saving approximately fifty to one hundred milliseconds per query, and can improve coherence for narrowly defined, repetitive tasks. However, these gains come at the expense of flexibility. Each policy update or regulatory change requires retraining to incorporate new information. Effective fine-tuning also demands large, diverse datasets to prevent overfitting, a condition that many insurance organizations struggle to meet due to limited labeled data. Fine-tuned models lack explicit source attribution, meaning that verifying outputs requires manual document review, and integrating new content involves repeating the fine-tuning cycle rather than simply adding documents to a retrieval corpus.

In practice, hybrid strategies often deliver the best balance between adaptability and precision. One common approach involves fine-tuning models to align them with domain-specific style, terminology, and communication conventions, while using RAG to provide up-to-date policy content and factual grounding. Another configuration uses fine-tuned lightweight models for narrowly scoped tasks such as entity extraction, complemented by a RAG-based architecture for open-ended reasoning and contextual question answering. For insurance claims processing, however, RAG generally represents the superior primary choice, combining interpretability, maintainability, and regulatory transparency. Fine-tuning remains valuable for auxiliary subtasks where explainability and adaptability are less critical, but for core decision-support applications, RAG's dynamic and auditable nature provides a more robust foundation.

### 2.9.3 Hybrid OCR Strategy Rationale

Insurance claims processing involves a wide variety of document sources, including digitally created PDFs, scanned paper documents, and hybrid files containing both text and images. Because no single extraction method performs optimally across all of these formats, hybrid strategies that adaptively select extraction techniques based on document characteristics offer the best balance between accuracy, speed, and cost.

Native PDF text extraction using tools such as PyMuPDF or PDFMiner reads text directly from the PDF's internal data structures, bypassing optical character recognition (OCR) entirely and thereby eliminating recognition errors. For well-formed digital PDFs, the error rate is effectively zero, compared to typical OCR error rates of one to ten percent. Moreover, native extraction operates between ten times and one hundred times faster than OCR, on the order of milliseconds rather than seconds per page, making it particularly advantageous for high-volume claims processing environments. However, this method fails completely for scanned or image-based PDFs, which lack embedded text. To address this, systems employ automatic quality assessment mechanisms that analyze metrics such as total character count, character distribution, and the presence of expected content elements. When these heuristics indicate that native extraction has yielded sparse or low-quality output, the system automatically triggers an OCR fallback, thus optimizing for speed whenever possible while maintaining accuracy when necessary.

For documents that are particularly difficult to process, such as degraded scans, handwritten forms, or irregular layouts, multi-engine ensemble approaches can further enhance reliability. Running multiple OCR engines in parallel, such as Tesseract, Google Cloud Vision API, and AWS Textract, allows the system to aggregate results through voting or confidence-based selection, often reducing error rates by 20 to 40 percent compared to single-engine solutions. However, this improvement comes at a cost: the computational and financial expenses scale with the number of engines employed. Because cloud OCR APIs typically charge around $1.50 per 1,000 pages, ensemble processing can quickly become costly when applied indiscriminately. Consequently, selective ensemble deployment represents a pragmatic compromise. Routine documents can be processed using a single, cost-effective engine such as Tesseract, while multi-engine ensembles are reserved for high-value claims or for cases where initial confidence scores fall below acceptable thresholds. This strategy maintains high accuracy where it matters most without incurring excessive operational costs.

Reliable quality assessment underpins the effectiveness of all these hybrid strategies. Extraction quality can be evaluated using several complementary criteria. Heuristic checks examine basic statistical properties such as character count, symbol distribution, and the presence of key policy identifiers or financial values. OCR confidence scores, available from Tesseract and most cloud-based OCR services, provide more granular information at the word or character level. Mean confidence thresholds, for example, an average confidence below 70 percent, can automatically trigger human review or reprocessing. Cross-validation between native and OCR-based outputs also serves as a powerful diagnostic tool: significant discrepancies between the two often indicate problematic PDFs, encoding errors, or OCR failures that warrant further inspection.

Through this layered approach, combining native extraction, OCR fallback, multi-engine ensembles, and dynamic quality assessment, insurance claims systems achieve a resilient and efficient text extraction pipeline that accommodates the full spectrum of document types encountered in real-world operations.

### 2.9.4 Dual Vectorstore Architecture Rationale

The rationale for the dual vectorstore architecture, discussed in detail in chapter 2.6.4, lies in the need to address the distinct lifecycles, security requirements, and usage patterns of different document categories. By maintaining separate databases for permanent policy documents and temporary customer uploads, the system ensures both operational efficiency and data protection.

### 2.9.5 Technology Selection Summary

Technology selection was guided by a balance among performance, cost, maintainability, integration complexity, and regulatory compliance. The final choices combine mature, production-proven technologies with state-of-the-art AI capabilities, achieving an equilibrium between innovation and reliability that aligns with the conservative nature of the insurance industry and its low tolerance for operational failures. Chapter 3 provides a detailed discussion of the architectural integration, data flows, API contracts, and deployment configurations that implement these design principles.

## 2.10 Chapter Summary

This chapter has surveyed the technologies and methodologies underpinning AI-powered insurance claims assessment. It began with a review of prior work in claims automation, tracing the evolution from rule-based expert systems and traditional machine learning approaches to contemporary methods that leverage Large Language Models (LLMs), Retrieval-Augmented Generation (RAG), and advanced optical character recognition (OCR) techniques.

The discussion of LLMs focused on the Transformer architecture and the GPT family of models, including GPT-3, GPT-3.5-turbo, GPT-4, and GPT-4 Vision. We examined their capabilities in natural language understanding, reasoning, and text generation, as well as their limitations in terms of hallucinations, interpretability, and bias. These challenges highlighted the necessity of grounding techniques and human oversight to ensure reliability and transparency in insurance applications.

RAG was presented as a key solution to issues of hallucination and knowledge cutoff inherent in standalone LLMs. Its architecture, comprising a knowledge base, retriever, and generator, was analyzed in depth, along with the role of dense vector embeddings and vector databases such as ChromaDB employing HNSW indexing. The advantages of RAG for insurance applications were emphasized, particularly in terms of accuracy, transparency, and maintainability, while challenges such as retrieval quality, chunking strategies, and metadata filtering were also discussed. Advanced retrieval techniques, including hypothetical document embeddings (HyDE), reranking, and recursive retrieval, were introduced as mechanisms for improving retrieval precision.

The chapter also examined OCR technologies, tracing their evolution from traditional computer vision methods to modern deep learning architectures such as Tesseract, LayoutLM, and Donut. It explored domain-specific challenges in processing insurance documents, including handwriting, mixed content, and complex layouts, and discussed preprocessing techniques such as binarization, deskewing, and dewarping. Hybrid extraction strategies combining native PDF parsing and OCR, as well as the emerging role of GPT-4 Vision in multimodal document understanding, were analyzed for their potential to enhance document intelligence in claims workflows.

Subsequent sections addressed semantic retrieval through vector embeddings, charting progress from early word embeddings such as Word2Vec and GloVe to contextualized embeddings produced by BERT and OpenAI's text-embedding-ada-002. Vector databases, dual vectorstore architectures, and document chunking strategies were discussed in the context of scalable, high-precision semantic search for insurance documents.

The LangChain framework was introduced as a unifying layer for orchestrating LLM-based applications, integrating components such as models, chains, retrievers, and memory systems. Its role in implementing RAG pipelines, managing prompts, maintaining conversational context, and supporting production deployment was analyzed, highlighting its utility for rapid development and system modularity.

The human-in-the-loop design paradigm was then examined, emphasizing the importance of human oversight in high-stakes and regulated domains. We discussed varying levels of automation, design patterns such as confidence-based escalation and explainable reasoning, and strategies for aligning AI systems with existing workflows. Principles of interface design were explored, focusing on information hierarchy, explainability, and cognitive load management. Evaluation metrics were also reviewed, including character and word error rates (CER/WER) for OCR, faithfulness measures for RAG, and latency and cost indicators for overall system performance. The chapter noted that explainability and auditability, achieved through RAG's explicit source attribution and structured human review, support compliance with key regulatory frameworks such as the GDPR and the Insurance Distribution Directive (IDD).

Finally, the chapter outlined the key criteria guiding technology selection, including model choice based on reasoning complexity, latency, cost, and context requirements; the trade-offs between RAG and fine-tuning; hybrid OCR strategies; and the rationale for adopting a dual vectorstore architecture.

Collectively, the technologies and methods reviewed in this chapter establish the foundation for the system design and implementation described in Chapter 3, which details the overall architecture, component integration, and workflow design. Chapter 4 then presents the empirical evaluation, covering OCR accuracy (measured via CER/WER and precision–recall–F1 metrics), RAG faithfulness, and system performance in terms of latency, cost, and throughput.

# Chapter 3

# System Architecture

## 3.1 Problem Definition and Objectives

### 3.1.1 Formal Problem Statement

Insurance claims processing suffers from inefficiencies rooted in manual workflows, inconsistent policy interpretation, and lengthy processing times. The central problem can be formalized as follows: given heterogeneous claim documents D (scanned forms, photographs, receipts, policy texts), a natural language query Q from a customer or adjuster, and a structured knowledge base K containing policy contracts and customer records, the system must produce an accurate, explainable answer A grounded in verifiable sources from K and relevant extractions from D, while satisfying regulatory transparency requirements, maintaining data privacy, and achieving interactive processing latencies suitable for production deployment.

The problem decomposes into five interconnected challenges, each presenting distinct technical obstacles. Document understanding requires extracting structured text from documents varying in quality, such as digital PDFs with embedded text, degraded scans exhibiting noise and skew, and handwritten forms with high inter-person variability, as well as in format, including continuous text paragraphs, structured tables with merged cells, checkboxes indicating coverage selections, and damage photographs requiring visual interpretation, and in language, such as Italian policy language, English technical terminology, or mixed-language claim descriptions. A single claim submission might include a digitally generated policy PDF requiring native text extraction, a scanned handwritten claim form demanding OCR with aggressive preprocessing, and damage photographs needing computer vision analysis. This heterogeneity necessitates hybrid extraction strategies that adaptively select methods based on document characteristics, combining native PDF parsing, which is fast and accurate for digital documents, with image-based OCR employing deskewing, contrast enhancement, and noise reduction, which is robust for degraded scans, while maintaining quality assessment throughout to flag uncertain extractions for human verification.

Semantic retrieval demands identifying relevant policy clauses despite substantial vocabulary mismatches between natural language queries and formal contract language. A customer asking "Is my windshield covered?" must retrieve policy sections discussing "automotive glass damage" or "cristalli," Italian for glass, where lexical overlap is minimal but semantic equivalence is clear. This requires dense embeddings mapping text to high-dimensional vectors where semantic similarity corresponds to geometric proximity, approximate nearest neighbor algorithms providing logarithmic search complexity over large policy corpora containing tens of thousands of chunks, and metadata-aware retrieval combining vector similarity with structured filters such as customer ID, coverage type, and policy version to ensure only relevant documents are considered.

Grounded generation synthesizes factually accurate answers by integrating retrieved policy clauses and OCR-extracted content while citing specific sources, a requirement that distinguishes insurance applications from general chatbots. The system must avoid hallucinations where plausible but incorrect coverage interpretations could lead to financial losses or customer disputes, maintain strict context conditioning ensuring answers derive exclusively from retrieved documents rather than parametric model knowledge, adapt tone and detail level for different audiences since customers require accessible explanations, adjusters need technical precision, and regulators demand audit trails, and structure outputs consistently in numbered sections or formatted text to support frontend rendering and human review workflows.

Policy validation and financial calculation integrate business logic with AI outputs, verifying that customers hold active policies covering the claimed damage types, applying coverage limits by taking the minimum between requested amounts and policy maximums, deducting fixed or percentage-based deductibles using exact decimal arithmetic to avoid floating-point rounding errors, computing co-payments when applicable, generating detailed calculation traces documenting each transformation with intermediate values and applied rules, and emitting alerts for exceptional conditions such as coverage limit exceedance requiring customer notification, missing mandatory documents like police reports for theft claims, high-value claims exceeding thresholds demanding senior review, or policy exclusions detected through substring matching in claim descriptions. These calculations must be deterministic, auditable, and compliant with insurance industry financial standards.

Privacy and compliance maintenance address regulatory obligations through architectural design rather than post-hoc additions. The system separates ephemeral customer uploads, including claim forms, receipts, and damage photos containing personally identifiable information, from persistent policy documents such as generic contracts without customer data, automatically deletes session-scoped temporary data upon session termination to satisfy GDPR's storage limitation principle and right-to-erasure requirements, provides explainable decisions through RAG's source attribution where every answer cites specific policy clauses supporting IDD mandates for transparency in automated decisions, logs all processing stages with pseudonymous customer identifiers while excluding personally identifiable information from logs to prevent inadvertent exposure, and supports human-in-the-loop oversight through confidence-based escalation that flags uncertain OCR extractions, low-similarity retrievals, or ambiguous policy interpretations for expert review.

### 3.1.2 System Objectives and Success Criteria

Five primary objectives guided system design. Accuracy ensures that policy interpretations align with contract language, OCR extractions reproduce document content faithfully, and financial calculations generate verifiable reimbursement amounts with complete audit trails. Transparency requires that answers cite specific policy clauses, calculation steps document intermediate values and applied rules, and confidence scores flag uncertain cases for review. Efficiency targets interactive query completion within ten seconds for policy questions and thirty seconds for document analysis, batch throughput sufficient for operational volumes, and economically sustainable API costs achieved through selective model routing and caching. Scalability demands accommodation of growing corpora without performance degradation, achieved through logarithmic retrieval latency enabled by approximate nearest neighbor indexing and stateless service architectures supporting horizontal scaling. Compliance safeguards customer privacy through session-scoped storage and automatic deletion, provides explainable decisions with source attribution satisfying GDPR and IDD requirements, and supports human oversight through confidence-based escalation and audit logging.

Success criteria establish quantitative thresholds. OCR accuracy targets character error rates below five percent for digital PDFs and fifteen percent for scans, word error rates below ten percent for print and twenty-five percent for handwriting, and precision, recall, and F1 scores above ninety percent for field extraction. RAG faithfulness requires source citations for all factual claims, hallucination rates below five percent, and answer relevance exceeding eighty percent. System performance targets end-to-end latency below ten seconds for interactive queries and thirty seconds for document uploads, throughput sufficient to handle claim surges, and per-claim API costs remaining within operational budgets.

Human–AI agreement should exceed seventy-five percent, with disagreement patterns analyzed to identify systematic weaknesses and guide model or process refinements. Chapter 4 evaluates these criteria empirically.

### 3.1.3 Solution Approach Overview

The solution integrates three core technological components described in Chapter 2: Retrieval-Augmented Generation for grounded policy question answering, hybrid OCR with advanced preprocessing for robust document understanding, and human-in-the-loop design patterns for confidence-based escalation and transparent decision support. The system architecture follows a service-oriented paradigm that separates concerns across layers, comprising a web backend exposing REST APIs for frontend interaction, orchestration services coordinating workflows across subsystems, core AI components implementing RAG retrieval and generation, OCR extraction, and claims validation logic, a data layer managing structured customer records in SQLite and vectorized policy embeddings in ChromaDB, and a React-based frontend offering conversational interfaces and progressive disclosure dashboards.

The dual vectorstore architecture, detailed in section 2.6.4, maintains distinct embeddings for permanent policy documents and ephemeral customer uploads, ensuring privacy-compliant session management and consistent performance. The hybrid OCR strategy, presented in section 2.9.3, routes digital PDFs to native text extraction for maximum speed and accuracy while applying image-based OCR with multi-stage preprocessing for scanned documents, selecting the optimal extraction method adaptively according to document characteristics. The tiered LLM selection approach, discussed in section 2.9.1, assigns simple extraction tasks to fast and cost-efficient models such as GPT-3.5-turbo while reserving complex reasoning and policy interpretation for GPT-4, achieving a balanced trade-off between accuracy and operational cost. These architectural principles translate the theoretical foundations of Chapter 2 into a concrete, production-ready implementation designed to support real-world insurance claims workflows efficiently and transparently.

## 3.2 System Architecture Overview

### 3.2.1 Component Topology

The system follows a layered architecture with clear separation of responsibilities across five distinct tiers, each abstracting complexity from the layers above while exposing well-defined interfaces. The presentation layer consists of a React-based web application providing chat interfaces for policy questions, document upload controls with drag-and-drop support, result dashboards displaying answers with progressive disclosure of OCR confidence and field comparisons, and health status indicators polling backend readiness. The frontend communicates exclusively via HTTP APIs with the backend, ensuring clean separation that enables independent deployment, version control, and scaling. This decoupling allows frontend updates such as UI improvements or new visualizations without backend modifications, and backend enhancements like new AI models or improved extraction algorithms without frontend changes, supporting parallel development and incremental deployment strategies.

The application layer implements a FastAPI server exposing REST endpoints for chat, which manages policy question answering with conversation context, upload-and-analyze, which handles OCR-augmented document queries, process-claim, which executes the full claim workflow including validation and reimbursement calculation, health checks verifying component readiness and configuration, vectorstore status for monitoring RAG initialization, and temporary document cleanup for session management. Endpoint handlers remain deliberately thin, focusing on HTTP-specific concerns such as request validation via Pydantic schemas ensuring type safety and mandatory field presence, response serialization converting Python objects to JSON with appropriate status codes, and error translation mapping technical exceptions like ValueError, ConnectionError, or TimeoutError to user-friendly messages with domain-appropriate HTTP responses. CORS configuration permits cross-origin requests during development while restricting origins in production. Business logic resides in orchestration services, maintaining handlers concise and easily maintainable so that endpoint updates require minimal code modification.

The service layer encapsulates workflow orchestration coordinating subsystem interactions while enforcing business rules and data flow contracts. The chat service executes the policy question workflow, extracting client IDs from conversation context through regex patterns and LLM-based extraction, querying SQLite for policy data and formatting results as structured facts, appending output formatting constraints specifying section numbering, citation style, and deductible application, invoking the full RAG pipeline including client validation, retrieval, and grounded reasoning, and updating conversation context to preserve coherence across turns. The document service manages OCR-augmented workflows with higher complexity, extracting identifiers from context, question text, and OCR content in prioritized order, matching customers in the database through fallback strategies such as ID match, name combination, or license plate when available, constructing enhanced queries merging OCR previews and policy context, invoking RAG with OCR grounding to cross-reference uploaded documents with contract clauses, comparing extracted fields to database values to flag discrepancies, and returning comprehensive responses containing summaries, field comparisons, and metadata for frontend visualization.

The initialization service coordinates cold-start procedures ensuring resource availability before serving requests. It verifies database existence, rebuilding from CSV sources if missing, validates the document corpus, triggering PDF parsing and GPT-4 Vision extraction when necessary, loads dataframes into memory with appropriate encodings, initializes RAG instances lazily with parallel warm-up across LLM clients to minimize startup time, constructs or reloads vectorstores while logging token counts and costs, and performs validation prompts to confirm system health. Utility services centralize reusable logic, including customer service providing ID extraction and normalization methods, matching service implementing schema-agnostic resolution strategies, and policy service formatting SQL results into structured policy fact blocks. This modular structure enhances maintainability, reusability, and testability by isolating concerns and minimizing code duplication.

The AI integration layer provides the core intelligence through two advanced subsystems exposing concise interfaces while encapsulating complex internal logic. The RAG subsystem manages retrieval-augmented generation using lazy-loaded LLM clients configured with deterministic parameters—GPT-4 for reasoning and GPT-3.5-turbo for fast extraction—alongside OpenAI's embeddings client supporting batch operations with retry and timeout handling. It maintains dual vectorstores with separate persistence directories and lifecycle policies, permanent for static contracts and temporary for session-based uploads, ensuring privacy compliance. LangChain retrieval chains orchestrate embedding, similarity search, and context construction, while conversation management tracks client identity and coverage context for multi-turn coherence. The OCR subsystem coordinates hybrid PDF and image extraction, iterating through PyMuPDF, PDFMiner, and PyPDF for digital text and applying a six-stage preprocessing pipeline for scanned images including grayscale conversion, deskewing, contrast enhancement via CLAHE, denoising, adaptive thresholding with Otsu or Sauvola methods, and morphological cleanup. It selects optimal page segmentation modes based on document type, supports bilingual Italian–English OCR through parallel inference, aggregates per-word confidences for quality scoring, and applies regex-based field extraction merging OCR with contextual data. Both subsystems expose unified methods for initialization, execution, document addition, and health reporting, abstracting extensive logic behind minimal interfaces.

The data layer ensures reliable persistence through three complementary mechanisms with distinct lifecycles. SQLite manages structured policy and customer data across normalized tables storing risk attestations, policy summaries, and optional coverage details, queried via parameterized SQL to prevent injection and accessed through pandas dataframes for ease of manipulation. The processed contract corpus, stored as TSV files, contains hierarchically structured policy text with metadata enabling semantic chunking, citation, and content categorization, built through offline parsing and GPT-4 Vision extraction for tables and glossaries. ChromaDB maintains dual vectorstores persisting embeddings, metadata, and HNSW indexes as binary and SQLite-backed files, allowing rapid loading of precomputed embeddings for permanent stores while generating temporary, session-scoped stores for uploaded documents. This combination of lightweight relational storage, editable text-based corpora, and high-performance vector retrieval underpins both scalability and compliance by balancing persistence, transparency, and privacy.

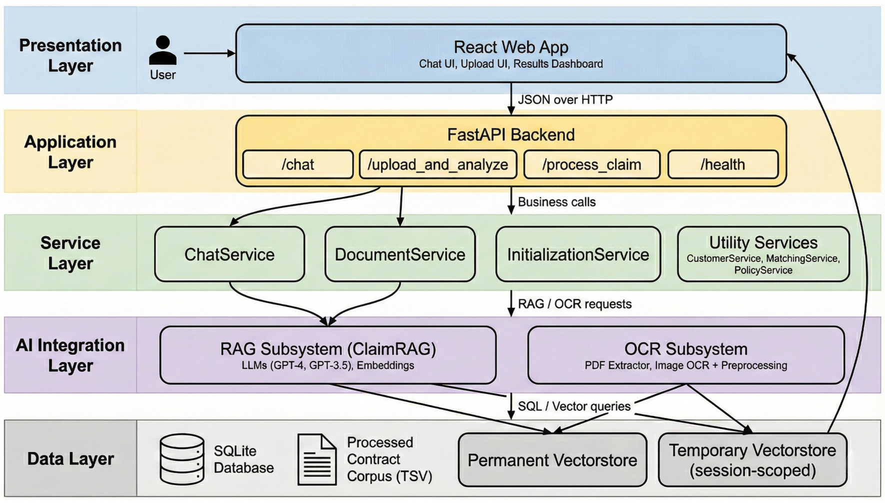

**Figure 3.1:** Five-tier layered architecture showing Presentation, Application, Service, AI Integration, and Data layers with data flow paths from user requests through processing to responses

### 3.2.2 Request Lifecycle and Data Flow

Policy question workflow: The user submits a question through the frontend, which transmits JSON data to the chat endpoint. The endpoint validates the schema and forwards the request to the chat service, which extracts the client ID from either the question text or the conversation context, enabling follow-up questions without requiring ID repetition. The service queries SQLite for the corresponding customer policy data and formats it into a structured policy facts block including coverage types, limits, deductibles, vehicle details, and premiums. This facts block, together with formatting rules, is appended to the user query. The RAG subsystem embeds the question into a 1536-dimensional vector, performs coverage type extraction through targeted retrieval using both metadata and semantic similarity, executes a second retrieval focused on the detected coverage, and constructs a comprehensive prompt integrating the retrieved passages, customer facts, and user question. The LLM, configured as GPT-4 with temperature zero and a five-hundred-token cap, generates a grounded response.

Post-processing standardizes bullet formatting, removes empty lines, and enforces consistent structure. Finally, the chat service updates conversation context to retain the client ID and coverage keywords, returning the cleaned and formatted answer to the frontend for rendering.

Document upload workflow: Users upload files with an accompanying question via multipart form-data sent to the upload-and-analyze endpoint. The endpoint validates file extensions (PDF, PNG, JPG, JPEG, TIFF, BMP) and size limits (default twenty megabytes), stores files temporarily, and initiates OCR processing. For PDFs, the system first attempts native text extraction via PyMuPDF, achieving zero error for embedded text and millisecond-level processing time. When extraction quality is low, determined by sparse text coverage below fifty percent, the pipeline automatically falls back to image-based OCR. For image files, a six-stage preprocessing pipeline—grayscale conversion, deskewing, contrast enhancement via CLAHE, denoising, adaptive thresholding with Otsu or Sauvola methods, and morphological cleanup—optimizes input for Tesseract, which performs recognition using a page segmentation mode selected according to document type (PSM 11 for sparse receipts, PSM 6 for uniform policies, PSM 3 for automotive forms). Low-confidence OCR outputs, below sixty percent mean confidence, trigger automatic fallback testing across alternative PSM modes (6, 7, 8, 11, 13) to select the most reliable result.

OCR results are passed to the document service, which extracts key identifiers from conversation context, question text, and OCR output such as cliente_id, nome, cognome, targa, and franchigia. It matches these against the database using prioritized resolution strategies—ID exact match, name combination, or license plate correlation—then formats corresponding database and policy blocks if a match is found. The system constructs an enhanced question combining OCR-derived text and policy context, invokes RAG with OCR grounding to cross-reference extracted information with contract clauses, compares OCR-extracted fields with database values to flag discrepancies, and returns a structured response including the generated answer, document summary with filename and confidence metrics, field-level comparison, and metadata for verification. The frontend renders this information using progressive disclosure to balance clarity and transparency, while temporary files are asynchronously deleted by background cleanup tasks to ensure privacy compliance.

### 3.2.3 Design Principles

Six design principles guided the architectural decisions. Separation of concerns decouples transport, orchestration, and domain logic, ensuring that endpoint handlers manage HTTP communication, services coordinate workflows, and AI components implement intelligence. This separation allows each layer to be tested, deployed, and scaled independently. Fail-safe defaults guarantee graceful degradation through automatic fallback mechanisms, such as switching to OCR when native PDF extraction fails, reverting to GPT-3.5-turbo when GPT-4 times out, or using pattern-based extraction when LLM inference becomes too costly. Lazy initialization defers the creation of computationally expensive resources like LLM clients and vectorstores until they are first needed, minimizing cold-start latency and optimizing startup efficiency. Privacy by design embeds data protection into the architecture through session-scoped temporary storage, strict separation of permanent and ephemeral embeddings, and systematic exclusion of personally identifiable information from logs. Explicit over implicit promotes traceable and auditable data flows by incorporating structured logging at every critical checkpoint, including client ID extraction, database queries, retrieval operations, LLM invocations, and financial calculations, enabling precise debugging and post-hoc audit reconstruction. Cost awareness constrains operational expenses by enforcing token usage limits, employing selective model routing, leveraging caching mechanisms, and integrating observability hooks that record token consumption and API costs for continuous monitoring and optimization.

## 3.3 Backend Architecture and API Design

### 3.3.1 Application Framework and Lifecycle Management

The backend employs FastAPI, chosen for its native async/await capabilities that enable efficient concurrent request handling, automatic OpenAPI documentation generation at the /docs endpoint, seamless integration with Pydantic for declarative schema validation, dependency injection facilitating clean separation of concerns, and compatibility with LangChain for AI orchestration. Uvicorn serves as the ASGI server on port 8001, configured with hot-reload in development and production-optimized settings using multiple workers and suppressed access logging to reduce overhead.

Application lifecycle management uses asynchronous context managers defining startup and shutdown handlers. During startup, the system ensures the existence of the permanent vectorstore directory (data/databases/chroma_db_permanent), cleans the temporary vectorstore directory (data/temp/chroma_db_temp) to remove residual session data, configures logging hierarchies with application modules at INFO level and third-party libraries at WARNING to reduce verbosity, and enables CORS middleware for the development frontend (http://localhost:8000), with stricter origin restrictions enforced in production environments.

The OCR processor initializes synchronously based on the OCR_MODE environment variable, which selects among three operational profiles: fast mode with a ten-megabyte limit and minimal preprocessing, standard mode with a twenty-megabyte limit and full six-stage preprocessing, and quality mode with a fifty-megabyte limit and exhaustive preprocessing including PSM fallback. If initialization fails, the processor is set to None, causing OCR endpoints to return HTTP 503 Service Unavailable while allowing non-OCR endpoints such as chat and health checks to remain fully functional, ensuring graceful degradation under partial system failure.

The RAG subsystem initializes asynchronously in a background task to prevent blocking health-check requests. The initialization service verifies the presence of essential resources, rebuilding the SQLite database from CSV files if absent, reconstructing the document corpus by triggering PDF parsing and GPT-4 Vision extraction if missing, and loading dataframes with UTF-8 encoding and tab-separated formatting. It then creates a ClaimRAG instance and executes parallel warm-up across the fast LLM, main LLM, and embeddings client while loading or building the permanent vectorstore. Warm-up latency typically ranges from eight to fifteen seconds when reusing existing vectorstores, or several minutes during the initial embedding phase on first deployment. Initialization errors are logged but do not halt application startup; affected endpoints temporarily return "initializing" messages or HTTP 503 responses until the process completes.

During shutdown, the RAG subsystem performs cleanup by closing temporary vectorstore connections, deleting temporary persistence directories through shutil.rmtree, nullifying in-memory attributes, and clearing caches and conversation context. The OCR processor releases internal database connections. A final utility function executes a comprehensive cleanup of temporary vectorstore data, ensuring that no customer uploads persist beyond the application lifecycle. This behavior enforces privacy-by-design principles, guaranteeing automatic deletion of ephemeral data even under abnormal termination conditions.

### 3.3.2 Static Asset Serving and Frontend Integration

The backend serves the production-built frontend as static assets mounted under the /static and /assets paths, with explicit routes defined for the favicon and logo. A catch-all route delivers frontend files or returns index.html to support client-side routing while explicitly excluding API paths such as /chat, /upload_and_analyze, and /health to prevent HTML responses from interfering with API consumers. During development, the frontend and backend operate on separate ports, 8000 and 8001 respectively, with CORS enabled to permit cross-origin communication. In production, static files are bundled together with the backend, allowing both the API and the user interface to be served from port 8001. This unified deployment model simplifies configuration and maintenance, offering a streamlined setup particularly suitable for small-to-medium insurance operations.

### 3.3.3 Endpoint Design and API Contracts

Six primary endpoints expose the system's capabilities through clearly defined request–response contracts. The health endpoint (GET /health) functions as a liveness and readiness probe, returning structured JSON with multiple diagnostic fields. The top-level status field reports "ready" when all components are initialized, "initializing" during warm-up, or "degraded" when any subsystem fails. The ready boolean is true only when both the RAG and OCR components are fully functional. A nested component_status object details the state of each subsystem (ocr_available, rag_initialized, database_connected, permanent_vectorstore_loaded), including boolean indicators and optional error messages for failed components.

The configuration object exposes non-sensitive runtime settings such as ocr_mode, max_file_size, log_level, and a boolean api_key_configured, while the iso_timestamp provides temporal correlation for logs. The frontend polls this endpoint every second during initialization, displaying progress indicators with contextual messages such as "Initializing RAG system..." or "Loading vectorstore..." until readiness is confirmed, preventing premature API calls that would otherwise return HTTP 503 responses.

The chat endpoint (POST /chat) accepts JSON payloads validated by Pydantic schemas requiring a non-empty question string (maximum length of ten thousand characters to prevent misuse) and an optional context dictionary preserving conversational continuity. The endpoint delegates to the chat service, which performs client ID extraction via regex and LLM fallback, retrieves policy data from SQLite, constructs a structured facts block, augments the question with formatting and context, and executes the full RAG pipeline (validation, retrieval, grounded reasoning). Responses include the generated answer string, updated context (tracking last_client_id and last_coverage for follow-up queries), and an optional processing_time_ms metric. Input validation errors return HTTP 400 with localized Italian messages such as "Specifica il cliente_id nella domanda" for direct user feedback.

Connection failures with OpenAI APIs trigger automatic RAG cleanup to prevent state corruption and return HTTP 500 with sanitized error messages. Requests exceeding three minutes are aborted with HTTP 504.

The upload-and-analyze endpoint (POST /upload_and_analyze) accepts multipart form-data containing a question field and an array of uploaded files. Validation enforces file type restrictions (pdf, png, jpg, jpeg, tiff, bmp) and a default twenty-megabyte size limit, rejecting invalid inputs with HTTP 400 and explicit reasons. Accepted files are stored temporarily using UUID-prefixed filenames before being processed by the OCR pipeline. The OCR output is passed to the document service, which extracts identifiers, matches customers, builds structured contexts, invokes RAG with OCR grounding, and performs field comparison. Responses include the generated answer, document_summary (filename, confidence, text preview), field_comparison (document and database values, match booleans for key fields), extracted_identifiers, database_info, and matching_strategy. The frontend visualizes this information with confidence badges (green for >90%, yellow for 60–90%, red for <60), tabular field comparisons, and collapsible text previews. Background cleanup tasks delete temporary files immediately after response delivery.

The process-claim endpoint (POST /process_claim) orchestrates the complete insurance claim lifecycle. It accepts structured fields including customer_id, date, location, description, damage_type, requested_amount, third_parties, documents, and files. Uploaded materials are validated and passed to the OCR processor's process_claim method, which performs extraction, pattern-based field recognition, merges results with user-submitted form data, constructs normalized ClaimData objects with decimal precision, queries policy data from the database, validates coverage, and calculates reimbursements with full traceability. Responses return customer_id, merged claim_data, coverage_status (covered, applicable_coverage, missing_coverage), missing_data, limits (coverage_limit, deductible, co_pay_percentage), estimated_reimbursement, alerts (coverage_limit_exceeded, missing_police_report, high_value_claim, policy_exclusion_detected), calculation_trace (including steps, reasoning, and timestamps), documents_processed, processing_time_ms, and a success boolean. This output supports internal audit dashboards and adjuster review workflows.

Auxiliary endpoints facilitate development and maintenance. The context endpoint (GET /context) exposes the current RAG conversation state for debugging. The vectorstore_status endpoint (GET /vectorstore_status) reports initialization progress, file paths, and memory statistics. The cleanup_temp_documents endpoint (POST /cleanup_temp_documents) triggers manual deletion of temporary session data. These endpoints are restricted to internal or administrative use, ensuring that production users access only core business functionalities.

## 3.4 Service Layer and Workflow Orchestration

### 3.4.1 Chat Service: Policy Question Answering

The chat service orchestrates RAG-based policy question answering through its process_chat_request method, which encapsulates the complete workflow from user input to grounded response. The method accepts four inputs: the question string containing the user's query, an optional conversation_context dictionary preserving dialogue state across turns, the rag_instance managing retrieval and generation, and the dataframe storing structured policy documents. It returns two outputs: the answer string generated by the LLM with document grounding, and the updated rag_instance reflecting the revised conversation context for subsequent queries.

The workflow begins with intent detection and slot management before proceeding to the standard RAG pipeline. The process executes through the following stages. First, intent classification and slot extraction analyze the user's question to determine the primary intent using LLM-based classification. Supported intents include compute_refund (calculating reimbursement amounts), determine_fault (assessing liability), coverage_validation (verifying policy coverage), document_consistency (checking document alignment), and policy_lookup (general policy questions). The system then extracts required slots from multiple sources following a priority hierarchy: conversation context (previously filled slots), question text (via pattern matching and LLM extraction), and OCR documents (when available). For each intent, specific slots are required: for example, compute_refund requires client_id, coverage_type, and damage_amount. If the system detects that the user wants to start a new case (through keywords like ’nuovo sinistro’ or LLM-based detection), it resets all stored slots to begin fresh.

Second, missing slot detection and clarification occurs if any required slots are missing after extraction. The system accumulates the user’s question in a pending_questions dictionary keyed by intent, stores the intent as pending_intent in conversation context, and generates a clarification message. The clarification uses the fast LLM (GPT-3.5-turbo) to create a natural-language prompt asking only for the missing information, ensuring the message is concise (maximum 15 words) and contextually appropriate. The system returns this clarification immediately without proceeding to RAG execution, enabling multi-turn information gathering where users provide required data incrementally.

Third, client ID extraction and name matching employs a multi-layered strategy that prioritizes existing context before invoking pattern recognition. The service first checks the conversation context for a previously identified client, enabling follow-up questions without re-specification. If unavailable, it applies regex-based matching to detect hexadecimal IDs (for example, "7e460f44"). Additionally, the system extracts nome and cognome from the question text using pattern matching (e.g., ’Marco Rossi’) and attempts to match them against the database to derive the client_id. If nome and cognome are provided in the current question, they take priority over any client_id from context, ensuring that explicit name-based queries are handled correctly. As a last resort, it performs an LLM-based extraction using the fast GPT-3.5-turbo model for natural-language formulations ("for my policy ending in 44"). Extracted IDs are validated against the database to ensure they correspond to actual customers, preventing queries about non-existent or inactive records.

Fourth, slot validation occurs once all required slots are present. The system validates them using business rules specific to each intent type. For compute_refund intent, it ensures damage amounts are positive numeric values, validates that coverage types match event types (e.g., ’garanzia cristalli’ covers ’vetro rotto’), and normalizes deductible overrides if provided. For coverage_validation, it verifies that both coverage type and event type are present and compatible. For determine_fault, it ensures event details contain sufficient description (minimum 10 characters). Validation failures return error messages in Italian, guiding users to correct their input.

Fifth, policy facts block construction queries the SQLite database through parameterized SQL joins across three tables: attestato_di_rischio_linked (containing identifiers and metadata), scheda_polizza_rca_linked (mandatory coverage data), and garanzie_opzionali_allianz_direct_linked (optional coverage information). The combined dataset is formatted as JSON containing cliente_id, coverage types (RCA, cristalli, furto, incendio), corresponding limits, deductibles, co-pay percentages, vehicle specifications, premium amounts, and claims history. This structured facts block provides comprehensive factual grounding to ensure contractually correct interpretation during generation.

Sixth, question enhancement augments the user query with formatting directives that enforce response consistency. These rules specify numbered sections, hyphen-based lists, exclusion of inline formatting such as Markdown or HTML, and restriction of citations to contractual clauses only. Shared coverage limit guidelines are also appended, ensuring that the model references policy text consistently across all responses and avoids stylistic variability that might confuse downstream consumers.

Seventh, refund payload computation occurs when the primary intent is compute_refund and all required slots are present. The system automatically computes the reimbursement amount using the compute_refund_payload method in the RAG instance. This method prioritizes data sources in the following order: (1) claim payload objects derived from OCR documents (highest priority, as they represent verified document extractions), (2) slot values extracted from the current question or conversation context. The computation considers damage amounts, deductibles, co-pay percentages, coverage limits, and coverage percentages, producing a structured payload that includes intermediate calculation steps and final reimbursement amount. This payload is stored in the filled slots and can be used by the RAG pipeline to provide precise financial answers.

Eighth, RAG execution invokes the run_full_pipeline_parallel method, which now accepts slots and conversation_context parameters in addition to the question and dataframe. These parameters enable the RAG subsystem to tailor its reasoning based on the detected intent and extracted information, potentially using pre-computed refund payloads for financial queries. The method performs client validation, executes database retrieval, extracts relevant coverage sections using targeted metadata-based search, and constructs a structured reasoning prompt combining retrieved passages, policy facts, user question, and formatting rules. The LLM (GPT-4, temperature set to zero, maximum five hundred tokens) generates the grounded answer asynchronously. The system logs retrieved passages, coverage matches, and the generated response for auditing and reproducibility.

Ninth, conversation context update analyzes the generated answer to extract any coverage keywords (cristalli, RCA, kasko, furto, incendio) or client identifiers, automatically persisting them into the session context. The system updates conversation state by storing filled slots in a slots dictionary organized by intent, enabling slot reuse across conversation turns. It also records the last client ID and coverage type for follow-up question handling. Additionally, if a refund payload was computed, it may be persisted in the conversation context for reference in subsequent queries. This mechanism enables seamless multi-turn dialogue where subsequent questions ("and what about theft?") inherit the prior context without requiring explicit re-entry of identifiers or coverage details.

Tenth, answer normalization standardizes the final output by converting Unicode bullets to hyphens, collapsing redundant line breaks, and trimming trailing whitespace. This ensures consistent formatting across sessions and mitigates stylistic variance introduced by LLM sampling.

Validation errors produce localized Italian messages to facilitate user understanding: "Specifica il cliente_id nella domanda" (no client ID extracted), "Cliente non trovato nel database" (invalid or missing customer record), "Nessuna copertura trovata" (no relevant policy coverage identified), and "Errore durante la generazione della risposta" (unexpected internal failure).

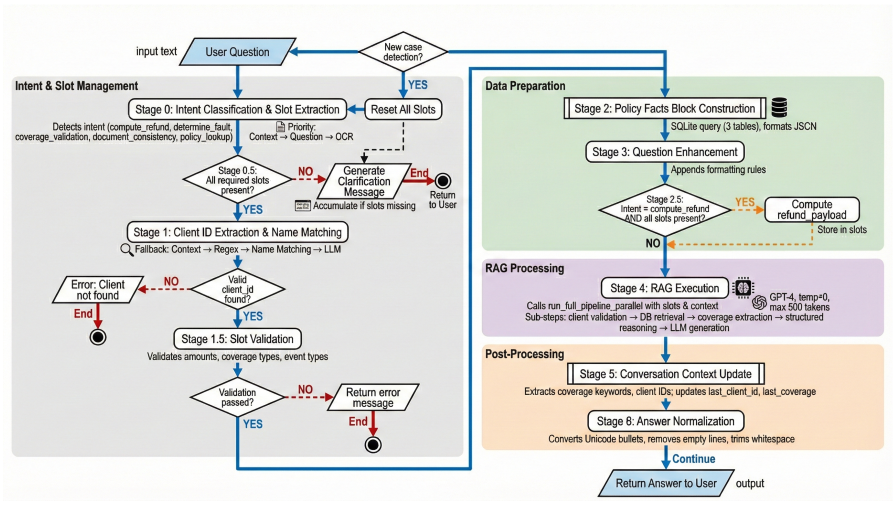
input text
User Question
New case detection?
YES
Intent & Slot Management
Stage 0: Intent Classification & Slot Extraction
Reset All Slots
Detects intent (compute_refund, determine_fault, coverage_validation, document_consistency, policy_lookup)
Priority: Context → Question → OCR
Stage 0.5: All required slots present?
NO
Generate Clarification Message
End
Return to User
Accumulate if slots missing
YES
Stage 1: Client ID Extraction & Name Matching
Fallback: Context → Regex → Name Matching → LLM
Valid client_id found?
NO
Error: Client not found
End
YES
Stage 1.5: Slot Validation
Validates amounts, coverage types, event types
Validation passed?
NO
Return error message
End
YES
Data Preparation
Stage 2: Policy Facts Block Construction
SQLite query (3 tables), formats JSCN
Stage 3: Question Enhancement
Appends formatting rules
Stage 2.5: Intent = compute_refund AND all slots present?
YES
Compute refund_payload
Store in slots
NO
RAG Processing
Stage 4: RAG Execution
GPT-4, temp=0, max 500 takens
Calls run_full_pipeline_parallel with slots & context
Sub-steps: client validation → DB retrieval → coverage extraction → structured reasoning → LLM generation
Post-Processing
Stage 5: Conversation Context Update
Extracts coverage keywords, client IDs; updates last_client_id, last_coverage
Stage 6: Answer Normalization
Converts Unicode bullets, removes empty lines, trims whitespace
Continue
Return Answer to User
output

**Figure 3.2:** Complete workflow from user question to answer, showing ten stages including intent classification, slot extraction, client ID matching, validation, RAG execution, and context update

## 3.4.2 Document Service: OCR-Augmented Analysis

The document service orchestrates OCR-augmented query answering through an intent-driven workflow that integrates document processing with slot management. The method accepts five inputs: the question string containing the user query, an array of OCR-processed documents (each including filename, extracted_text, and confidence_score), an optional conversation_context dictionary preserving dialogue state, the rag_instance handling retrieval and generation, and the dataframe containing structured policy data. It returns six outputs: the generated answer string grounded in OCR and database context, a field_comparison object identifying mismatches between extracted and database values, an extracted_identifiers object summarizing all detected identifiers, a database_info object containing matched customer records, a matching_strategy string describing the identification method used ("cliente_id", "nome_cognome", "targa", or null if unmatched), and a document_summary array with metadata for each processed file.

The process executes through the following stages. First, new case detection and payload initialization checks if the user wants to start a new case using the same detection mechanism as the chat service. If detected, it resets all stored slots and clears any existing claim payload. The system then initializes or loads a structured ClaimPayload object from conversation context, which aggregates extracted information across multiple conversation turns. This payload object maintains field values with confidence scores and document provenance, enabling incremental information gathering where users can provide additional documents or details in subsequent turns.

Second, document usability classification separates documents into two categories before processing: usable documents (those with successful OCR extraction and non-empty text) and unreadable documents (those that failed OCR or produced no extractable text). If all documents are unreadable, the system immediately returns an error message without proceeding. If some documents are unreadable, the system continues processing with usable documents but flags the failures for inclusion in the final response, ensuring partial success scenarios are handled gracefully.

Third, intent classification and slot extraction from documents applies the same intent detection and slot extraction workflow as the chat service, but with OCR documents included as an additional source. The analyse_request function receives the question text and the list of usable documents, extracting slots from both sources. Document-extracted fields (such as damage amounts, coverage types, or client identifiers) are aggregated into the claim payload object, with higher-confidence extractions taking precedence when multiple sources provide conflicting values. If required slots are missing, the system generates clarification messages and returns immediately, just as in the chat workflow.

Fourth, slot validation occurs once all required slots are present. The system validates them using the same business rules as the chat service, ensuring coverage types match event types, amounts are valid numeric values, and fault determination requests include sufficient detail. Validation failures return error messages without proceeding to RAG execution.

Fifth, identifier extraction and aggregation executes after intent classification and slot extraction, extracting client identifiers from multiple sources following a priority hierarchy: conversation context (previously identified client), question text (via pattern matching), and OCR documents (via field extraction). It targets key entities including cliente_id (hexadecimal IDs using /[0-9a-f]6,8/i), nome and cognome (via keyword matching and name heuristics), targa (license plate patterns /[A-Z]23[A-Z]2/), and franchigia (monetary amounts via currency-aware regex). The extraction process aggregates fields from all usable documents, combining nome, cognome, targa, and franchigia values. The claim payload manager updates the payload object with any newly extracted fields, maintaining confidence scores and document sources for auditability. Context-provided identifiers take precedence over OCR-derived ones, which in turn override those extracted from the question, maintaining a reliability hierarchy that favors previously validated information.

Sixth, database matching applies cascading strategies to locate customer records. The system first attempts an exact ID match; if successful, it immediately returns results with matching_strategy set to "cliente_id". If unsuccessful, it attempts a combined name match using normalized name-surname concatenation ("nome_cognome" strategy), followed by a license plate match ("targa" strategy) for automobile-related queries. If all strategies fail, the system continues gracefully with null matching strategy, allowing downstream RAG execution with placeholder values instead of halting the workflow.

Seventh, database information formatting constructs structured context blocks from matched records, including identifiers, policy metadata, coverage limits, and historical claims. It reuses the same SQL joins as the chat service (attestato_di_rischio_linked, scheda_polizza_rca_linked, garanzie_opzionali_allianz_direct_linked), ensuring consistent policy representation. If no match is found, the block populates "Non disponibile" placeholders to maintain prompt integrity for the RAG subsystem while signaling incomplete data. Eighth, enhanced question construction builds an enhanced question that integrates OCR results, policy context, and database information. The system concatenates extracted text snippets from usable documents—truncated to one thousand characters per document for efficiency—formatted with normalized bullets for consistency, separated by visual delimiters and appended to the user's question along with database context and formatting rules. If unreadable documents exist, their filenames and error messages are appended to the prompt for reference, though they are explicitly marked as unavailable to the model. The enhanced question also includes database information blocks and policy facts, creating a comprehensive context for RAG reasoning. The question is constructed with the same formatting rules as the chat service, ensuring consistent output structure. Explicit OCR usage instructions guide the LLM to rely on extracted content for grounding, enabling precise cross-referencing between scanned document data and contractual clauses.

Ninth, RAG execution with intent-aware processing performs the main reasoning phase. The system invokes the RAG pipeline with the enhanced question, passing slots and conversation_context parameters. For valid customer IDs, it executes the standard pipeline—client validation, policy data retrieval, coverage extraction via metadata-filtered similarity search, structured reasoning, and deterministic GPT-4 generation (temperature zero, maximum five hundred tokens). When customer matching fails, it gracefully degrades to general policy reasoning with placeholders, still generating contextually useful responses grounded in available policy data and extracted text. In both cases, the slots parameter enables intent-specific reasoning, and the conversation context provides access to the claim payload for financial computations or validation checks.

Tenth, field comparison evaluates extracted identifiers against authoritative database values, creating triplets of document_value, database_value, and match boolean flags. The field comparison now includes franchise (franchigia) comparison when available, checking both document-extracted and database values for consistency. Discrepancies are categorized as OCR recognition errors, record mismatches, or potential data inconsistencies. The frontend visualizes comparison results using a traffic-light scheme: green for matches, yellow for minor deviations, and red for critical conflicts that require manual review.

Eleventh, unreadable document notification appends a notification to the answer if any documents were unreadable, explaining which documents could not be processed and why, ensuring users are informed about partial processing failures. This notification is generated using fallback message builders that create contextually appropriate Italian messages.

Twelfth, result packaging aggregates all artifacts into a unified response structure. The return value includes the generated answer, field comparison results (comparing extracted identifiers against database values), extracted identifiers summary, database information for matched customers, matching strategy used (cliente_id, nome_cognome, or targa), and the updated RAG instance. Each document's summary includes filename, confidence_score, text_preview, and processing_time_ms, while accompanying objects include field_comparison, extracted_identifiers, database_info, and matching_strategy. This comprehensive structure enables the frontend to render interactive views with confidence indicators, collapsible OCR previews, side-by-side comparison tables, and customer profile cards, facilitating transparent human verification and end-to-end traceability.

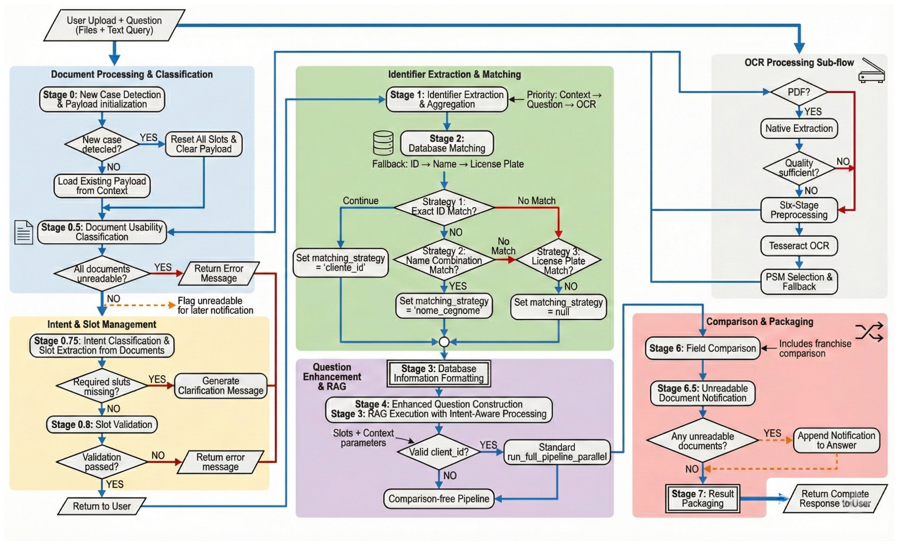
User Upload + Question (Files + Text Query)
Document Processing & Classification
Stage 0: New Case Detection & Payload initialization
New case detecled?
YES
Reset All Slots & Clear Payload
NO
Load Existing Payload from Context
Stage 0.5: Document Usability Classification
All documents unreadable?
YES
Return Error Message
NO
Flag unreadable for later notification
Intent & Slot Management
Stage 0.75: Intent Classification & Slot Extraction from Documents
Required sluts missing?
YES
Generate Clarification Message
NO
Stage 0.8: Slot Validation
Validation passed?
NO
Return error message
YES
Return to User
Identifier Extraction & Matching
Stage 1: Identifier Extraction & Aggregation
Priority: Context → Question → OCR
Stage 2: Database Matching
Fallback: ID → Name → License Plate
Continue
Strategy 1: Exact ID Match?
No Match
NO
Set matching_strategy = 'cliente_id'
Strategy 2: Name Combination Match?
No Match
Strategy 3: License Plate Match?
YES
Set matching_strategy = 'nome_cegnome'
NO
Set matching_strategy = null
Question Enhancement & RAG
Stage 3: Database Information Formatting
Stage 4: Enhanced Question Construction
Stage 3: RAG Execution with Intent-Aware Processing
Slots + Context parameters
Valid client_id?
YES
Standard run_full_pipeline_parallel
NO
Comparison-free Pipeline
OCR Processing Sub-flow
PDF?
YES
Native Extraction
Quality sufficient?
NO
NO
Stx-Stage Preprocessing
Tesseract OCR
PSM Selection & Fallback
Comparison & Packaging
Stage 6: Field Comparison
Includes franchise comparison
Stage 6.5: Unreadable Document Notification
Any unreadable documents?
YES
Append Notification to Answer
NO
Stage 7: Result Packaging
Return Complete Response to User

**Figure 3.3:** Twelve-stage workflow for OCR-augmented queries, including document classification, intent detection, identifier extraction, database matching, RAG execution, and field comparison

## 3.4.3 Initialization and Utility Services

The initialization service bootstraps the system through its initialize_system method, executing four idempotent verification steps that ensure all resources exist and are correctly configured before serving user requests.

First, database existence verification checks for the presence of assicurazioni.db. If the file is missing, it rebuilds the database from three canonical CSV sources: attestato_di_rischio_linked (identifiers and metadata), garanzie_opzionali_allianz_direct_linked (optional coverage details), and scheda_polizza_rca_linked (mandatory coverage). The initialization process defines tables with appropriate data types, primary keys, and indexes to support efficient querying. This ensures consistent schema reconstruction across deployments while maintaining referential integrity between the core entities.

Second, document corpus verification ensures the existence of the processed policy file documento_assicurativo.txt. If absent, the system invokes the CGA parser to rebuild it by parsing raw policy PDFs using the Fitz library for text extraction, invoking GPT-4 Vision for glossary and table recognition, and merging outputs into a structured TSV file. Each record contains section, subsection, title, text, page, and type columns, preserving document hierarchy and enabling semantic chunking during retrieval. This process transforms unstructured policy PDFs into machine-readable corpora suitable for vectorization and downstream RAG operations.

Third, dataframe loading reads the TSV corpus into a pandas dataframe using explicit configuration (tab separator, UTF-8 encoding, enforced column schema). Missing values are filled with nulls to ensure downstream compatibility. The dataframe is cached in memory to provide low-latency access during RAG retrieval, supporting metadata filtering and search operations without repeated disk reads. This memory-resident structure becomes the central reference for document queries and citation lookups.

Fourth, RAG initialization constructs a ClaimRAG instance and performs parallel warm-up using asynchronous execution (asyncio.gather). Three components initialize concurrently: the fast LLM for lightweight extraction tasks, the main LLM for reasoning and generation, and the embeddings client for document retrieval. The service builds or loads the permanent vectorstore containing precomputed embeddings, metadata, and HNSW indexes stored in a combination of binary and SQLite formats. It validates readiness by issuing test prompts and automatically reinitializes any stale or unresponsive components. Warm-up typically completes within eight to fifteen seconds if vectorstores exist or several minutes during first-time embedding.

Complementing the initialization process, additional utility services provide specialized functionality for intent-driven workflows and structured data management. The slot manager service centralizes intent detection and slot extraction logic, enabling the system to classify user requests into specific action types (compute_refund, determine_fault, coverage_validation, document_consistency, policy_lookup) and extract required information fields from multiple sources. It implements a priority hierarchy for slot extraction: conversation context (previously filled slots that can be reused), question text (via pattern matching and LLM-based extraction), and OCR documents (via structured field extraction).

The service maintains a lexicon of coverage keywords that maps natural language terms to standardized coverage types (e.g., 'cristalli', 'vetro', 'lunotto' map to 'garanzia cristalli'), enabling robust coverage type inference. It also implements slot collection logic that aggregates values from all sources, with higher-confidence extractions taking precedence when conflicts occur. The validation service enforces business rules specific to each intent type, ensuring that extracted slots meet quality and consistency requirements.

For compute_refund intent, it validates that damage amounts are positive numeric values, normalizes monetary amounts to two-decimal precision, and verifies that coverage types are compatible with event types (e.g., ensuring 'garanzia cristalli' covers events like 'vetro rotto'). For coverage_validation intent, it ensures both coverage type and event type are present and compatible. For determine_fault intent, it verifies that event details contain sufficient description (minimum 10 characters) to enable meaningful fault assessment. The payload service manages structured ClaimPayload objects that aggregate extracted information across multiple conversation turns, maintaining field values with confidence scores and document provenance. It implements a priority-based field assignment system where higher-confidence extractions replace lower-confidence ones, ensuring that document-derived information (typically higher confidence) takes precedence over question-extracted values. The payload object maintains separate sections for identifiers (cliente_id, policy_id, targa), coverage information (coverage_type, deductible, co_pay_percentage, coverage_percentage, limits), event details (date, location, description), financial information (damage amounts), and document metadata. This structured representation enables incremental information gathering where users can provide additional documents or details in subsequent conversation turns, with the system automatically merging new information into the existing payload.

The customer service provides field extraction and normalization utilities used throughout the system, including by the slot manager for extracting identifiers from text. It defines regex patterns for common insurance fields—cliente_id (/[0-9a-f]6,8/i), targa (/[A-Z]23[A-Z]2/), nome and cognome via keyword-based heuristics, and franchigia through currency-aware expressions. It aggregates fields from multiple sources, normalizes strings, converts bullets to hyphens, cleans irregular responses, and enforces the extraction priority order (context → question → OCR). The service also provides text cleaning utilities that normalize bullet formatting, remove empty lines, and standardize response structure.

The matching service resolves customer identities using prioritized strategies: exact ID match, combined name–surname match, and license plate match. It accommodates schema variations by performing adaptive column detection and returns a matched database row along with a matching_strategy indicator or an empty dictionary if unresolved. This service is used by both the chat service (for name-based client identification) and the document service (for matching extracted identifiers against the database), enabling the system to report how customer identification was achieved.

Finally, the policy service formats retrieved database rows into structured information blocks containing identifiers, customer details, vehicle plate, insurance class, and claims history. It queries for policy facts—coverage types, limits, deductibles—via SQL joins across the three primary tables. Missing data are replaced with "Non disponibile" placeholders to preserve structural consistency. Schema-agnostic column detection ensures compatibility with heterogeneous datasets, allowing flexible ingestion of new insurance schemas without manual reconfiguration.

## 3.5 Retrieval-Augmented Generation Subsystem

### 3.5.1 Architecture and Component Responsibilities

The RAG subsystem, implemented in the ClaimRAG class, orchestrates the complete retrieval-augmented generation workflow for grounded policy question answering. It manages three principal resources: large language models responsible for reasoning and information extraction, an embeddings model supporting semantic retrieval, and dual vectorstores handling permanent contract documents and temporary OCR uploads. The subsystem exposes a concise public interface including methods for parallel warm-up of models, building and loading vectorstores from document corpora, executing the full RAG pipeline for policy queries, adding or clearing temporary OCR-derived documents, and querying vectorstore status for operational observability. Internally, it employs lazy initialization to defer expensive resource creation until first use, document and token count caching to optimize performance, asynchronous concurrency across all major operations, conversation context tracking to preserve continuity in follow-up questions, and structured logging capturing latency breakdowns and component-level execution times for performance monitoring.

The subsystem maintains strict persistence separation between its two vectorstores to enforce data lifecycle and privacy guarantees. The permanent vectorstore, located at data/databases/chroma_db_permanent, stores embeddings derived from official contract documents. It persists across application restarts, allowing the embedded corpus to be reused without recomputation unless new policy documents are added or existing ones are updated. This ensures both efficiency and consistency of retrieval results across sessions.

The temporary vectorstore, located at data/temp/chroma_db_temp, stores embeddings generated from customer-uploaded OCR documents. It is session-scoped and automatically deleted during both startup and shutdown procedures, preventing any residual personal data from persisting beyond its intended use. This isolation guarantees that ephemeral, customer-specific data and persistent, institution-wide policy data remain entirely separate.

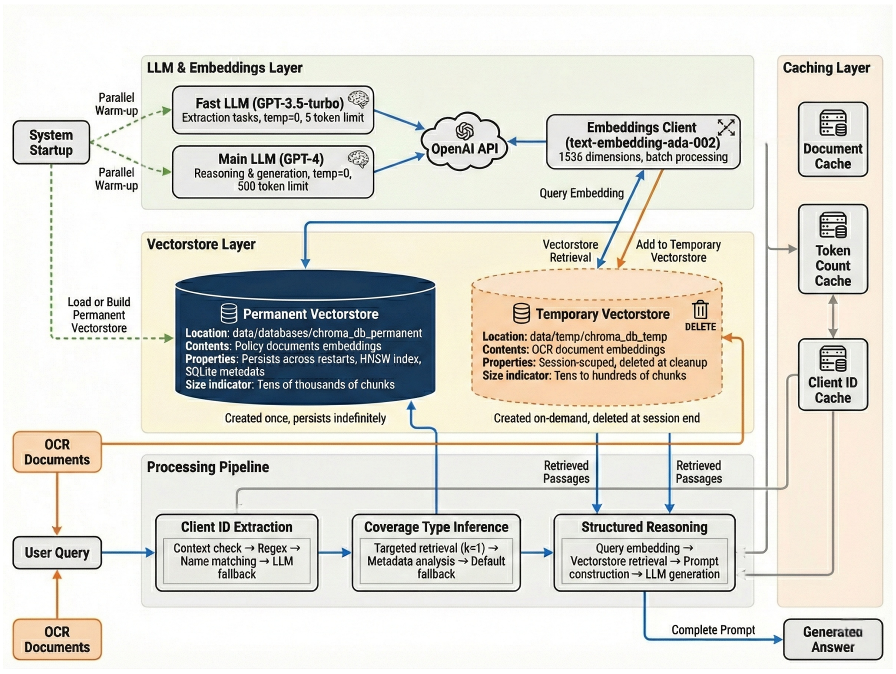
LLM & Embeddings Layer
Caching Layer
System Startup
Parallel Warm-up
Fast LLM (GPT-3.5-turbo)
Extraction tasks, temp=0, 5 token limit
Main LLM (GPT-4)
Reasoning & generation, temp=0, 500 token limit
OpenAI API
Embeddings Client (text-embedding-ada-002)
1536 dimensions, batch processing
Query Embedding
Document Cache
Token Count Cache
Client ID Cache
Vectorstore Layer
Vectorstore Retrieval
Add to Temporary Vectorstore
Load or Build Permanent Vectorstore
Permanent Vectorstore
Location: data/databases/chroma_db_permanent
Contents: Policy documents embeddings
Properties: Persists across restarts, HNSW index, SQLite metedats
Size indicator: Tens of thousands of chunks
Created once, persists indefinitely
Temporary Vectorstore
DELETE
Location: data/temp/chroma_db_temp
Contents: OCR document embeddings
Properties: Session-scoped, deleted at cleanup
Size indicator: Tens to hundreds of chunks
Created on-demand, deleted at session end
OCR Documents
Processing Pipeline
Retrieved Passages
User Query
Client ID Extraction
Context check → Regex → Name matching → LLM fallback
Coverage Type Inference
Targeted retrieval (k=1) → Metadata analysis → Default fallback
Structured Reasoning
Query embedding → Vectorstore retrieval → Prompt construction → LLM generation
Complete Prompt
Generated Answer

**Figure 3.4:** Internal RAG components showing LLM clients, embeddings, dual vectorstores, and pipeline stages (client ID extraction, coverage inference, structured reasoning) with data flows

## 3.5.2 Lazy Initialization and Parallel Warm-Up

All expensive resources in the subsystem implement lazy initialization through Python property decorators, ensuring that objects are instantiated only when first accessed. This design minimizes startup latency and conserves memory by avoiding unnecessary resource creation. The embedding model lazily instantiates an OpenAI Embeddings client configured for one-thousand-token chunk size, three automatic retries, and a one-hundred-twenty-second timeout. The main LLM property initializes a ChatOpenAI GPT-4 client configured with zero temperature for deterministic outputs, a ninety-second timeout, and a five-hundred-token generation limit suitable for reasoning tasks. The fast LLM property creates a GPT-3.5-turbo client with zero temperature, a thirty-second timeout, and a five-token limit optimized for rapid extraction operations. Because these clients establish connections only when invoked, the application achieves near-instant startup even while asynchronous warm-up continues in the background. Selective initialization further optimizes resource usage: components remain inactive unless explicitly required, such as the temporary vectorstore, which is created only when OCR workflows execute, minimizing memory footprint and avoiding redundant API sessions.

Vectorstores follow the same deferred-initialization principle. The permanent vectorstore loads within the build_vectorstore_async routine, first verifying the presence of its persistence directory. If an existing database is found, it loads the HNSW index and SQLite metadata directly; if absent, it embeds the document corpus using Chroma.from_documents, persists the resulting vectors and metadata for reuse, and logs the operation. The temporary vectorstore initializes dynamically when OCR documents are added through add_temp_documents, generating a session-scoped Chroma instance tied to the current workflow. Each initialization event records detailed logs including status (loading versus embedding), token counts, and elapsed time, providing transparency for performance and cost monitoring.

Warm-up executes asynchronously across three parallel tasks coordinated through asyncio.gather: a fast LLM check sending a single-token test ("ID"), a main LLM check using a short validation prompt ("Test"), and an embeddings check embedding a test query. Running these in parallel reduces warm-up duration from the cumulative latency of all components to the maximum latency among them. If a dataframe is provided, vectorstore initialization occurs after model warm-up to preserve dependency order, since embeddings rely on initialized clients. The subsystem implements graceful error handling: if a component fails during warm-up, it retries upon first use without affecting other successfully initialized components.

A global warm-up flag enforces idempotence, ensuring that initialization routines run only once per session regardless of concurrent requests.

### 3.5.3 Client ID Extraction with Context Tracking

Client ID extraction follows a multi-layered strategy designed to maximize speed, accuracy, and cost efficiency. The _extract_cliente_id method executes this process through prioritized sequential checks that escalate in computational complexity only as needed.

The first stage performs follow-up detection using lightweight heuristics to recognize when a user question implicitly refers to a previously identified client. The method searches for contextual keywords such as "questo cliente," "il cliente," "ha avuto," or "quanti," combined with the absence of explicit ID patterns. When this condition is met, it reuses the _last_client_id from conversation context, incurring zero latency and no API cost. This mechanism supports natural multi-turn dialogues without requiring the user to repeat the client ID in every question.

If the query is not a follow-up, the second stage applies regex-based pattern matching to detect explicit identifiers, typically six or more alphanumeric characters containing at least one letter and one digit (for example, "7e460f44"). Regex evaluation executes locally and instantly; when a match is found, the method returns immediately, bypassing further computation.

When both heuristic reuse and pattern matching fail, the third stage invokes LLM-based extraction using the fast GPT-3.5-turbo model. A minimal prompt—“extract only client ID”—guides the model to output the identifier with no extraneous text. The response is validated by the same regex used in the previous stage to ensure structural correctness. Results are cached by question text, so identical or paraphrased queries retrieve previously extracted IDs without re-invoking the LLM.

A fallback mechanism ensures graceful degradation. If no valid identifier is found after all steps, the system checks for an ID stored in conversation context before raising a ValueError, returning an Italian message prompting the user to specify the client ID explicitly (“Specifica il cliente_id nella domanda”).

After successful extraction, the subsystem updates conversation state, recording _last_client_id, refreshing _conversation_context, and incrementing a question counter for structured logging and dialogue tracking.

This tiered approach achieves an optimal balance:

– Accuracy through hybrid detection combining rule-based precision with LLM semantic understanding,

– Latency efficiency by prioritizing instant context reuse and regex matching before invoking an external model, and

– Cost control through response caching and selective use of the low-cost LLM only when necessary.

### 3.5.4 Coverage Type Inference and Targeted Retrieval

Coverage inference enhances retrieval precision by determining the most relevant insurance coverage category before executing the main retrieval phase. The extract_coverage_type method accomplishes this through a lightweight, low-latency embedding query that retrieves the single most semantically similar document to the user’s question. It then examines document metadata—particularly the section and subsection fields—for coverage-related keywords such as “garanzia RCA,” “garanzia cristalli,” “garanzia furto,” “garanzia incendio,” and “garanzia eventi naturali.” If no match is found within metadata, the method performs a secondary search directly within the document page_content, scanning for the same set of keywords to infer coverage context. When both checks fail, the system defaults to “garanzia RCA,” ensuring that the pipeline always continues with a valid coverage reference rather than interrupting processing.

This early-stage classification precedes structured_reasoning_parallel, allowing subsequent retrieval queries to be coverage-targeted. For instance, a question concerning glass repair deductibles triggers a focused retrieval instruction such as “Mostra solo il paragrafo che descrive la garanzia cristalli”, thereby isolating the relevant section instead of mixing unrelated coverages like theft or fire.

Operationally, coverage inference is highly efficient: it requires only a single embedding computation and a single vectorstore query (k=1), yielding typical latencies between one hundred and three hundred milliseconds—negligible compared to full pipeline execution time. In cases of retrieval or classification failure, the fallback to the default RCA coverage guarantees resilience, maintaining full workflow continuity with only minor precision degradation.

By constraining retrieval scope early in the process, this method significantly improves downstream reasoning accuracy, reduces hallucination risks, and accelerates policy interpretation by filtering the search space to the most relevant coverage subset.

### 3.5.5 Structured Reasoning and Prompt Construction

The structured_reasoning_parallel method coordinates the two critical phases of grounded question answering—targeted retrieval and large language model reasoning—within a unified, asynchronous workflow. It begins by constructing a coverage-specific Italian query that explicitly focuses retrieval on the relevant policy section, for example "Mostra solo il paragrafo che descrive la garanzia cristalli." This targeted phrasing prevents irrelevant passages from entering the reasoning context. The method then invokes the RAG chain's ask function, which queries the permanent vectorstore using the question embedding and retrieves the most semantically aligned passages. These grounding documents, typically returned within two hundred to five hundred milliseconds, are concatenated into a coherent text block for inclusion in the LLM prompt. The prompt template now follows a five-section structure that incorporates intent-specific instructions and extracted slot information, ensuring completeness, interpretability, and consistent formatting across responses while adapting to different user intents. The first section, "Dati cliente," presents the customer's structured policy information in JSON format, including fields such as cliente_id, coverage history, policy types, limits, deductibles, and co-pay percentages. The second section, "Dati confermati dalla richiesta" (Confirmed Data from Request), embeds the extracted slots as a JSON object, including client_id, coverage_type, damage_amount, intent, and any computed refund_payload. This section provides the model with structured information that has already been validated and normalized, enabling it to focus on reasoning rather than extraction. When a refund_payload is present (for compute_refund intent), it contains pre-computed financial calculations including intermediate steps, final reimbursement amount, and applied rules, allowing the model to present results without recalculating. The third section, "Documenti di contratto," embeds the retrieved grounding passages, preserving source structure for traceability. The fourth section, "Domanda," contains the user's natural language question, possibly enriched with conversational context or OCR-derived excerpts.

The final section, "ISTRUZIONI PER LA RISPOSTA" (Answer Instructions), now provides intent-specific guidance rather than generic formatting rules. For compute_refund intent with a refund_payload, the instructions direct the model to use the pre-computed payload without recalculating numbers, present the net reimbursement in a discursive sentence, list each calculation step from the payload in natural language (e.g., "Base dopo franchigia: 1.200,00 - 500,00 = 700,00"), and reference deductible, co-pay, and coverage limits only from values present in the payload or policy facts. If the refund_payload is missing, the instructions direct the model to declare that calculation is unavailable and indicate which data (damage amount, deductible, co-pay, coverage limit) are needed, without estimating amounts. For determine_fault intent, the instructions require synthesizing the event dynamics, indicating whether fault is present, absent, or information is unavailable, and citing any relevant notes from historical claims. For coverage_validation intent, the instructions require indicating whether the coverage applies to the requested event, explaining the reasoning by citing relevant policy clauses, and highlighting any conditions or exclusions. For policy_lookup intent (the default), the instructions require providing a comprehensive answer that summarizes relevant coverages and responds exhaustively to the question. All intent-specific instructions are followed by the standard formatting rules (numbered sections, lists, etc.) and explicit prohibitions against inventing examples or amounts.

The prompt construction process also incorporates slot-based coverage type selection. While the system typically extracts coverage type through retrieval-based inference (querying the vectorstore for the most semantically similar document and examining its metadata), if a coverage_type is present in the slots (extracted from the user's question or OCR documents), it takes precedence over the retrieval-based result. This ensures that explicit user specifications are respected, improving accuracy for cases where users directly state their coverage type.

The LLM invocation employs GPT-4 with zero temperature for deterministic outputs, a five-hundred-token cap to control verbosity and cost, and asynchronous execution enabling concurrent reasoning when multiple queries run in parallel. The method now accepts a slots parameter that influences both prompt construction (through the "Dati confermati" section) and instruction selection (through intent-specific guidance), enabling the model to provide more targeted and accurate responses based on the detected user intent and extracted information. Integrated observability hooks record token usage, execution time, and estimated API costs for each call, facilitating operational monitoring and optimization. Overall, the method achieves end-to-end reasoning latency of approximately three to eight seconds depending on passage length and model load, well within interactive thresholds for production environments. Upon completion, it returns the fully grounded LLM response, marking the final stage of the reasoning process before normalization and frontend delivery.

### 3.5.6 Conversation Context and Follow-Up Question Handling

Stateful conversation tracking sustains coherent multi-turn dialogues through multiple persistent internal attributes that maintain context across user interactions within a session. The conversation context has been extended beyond simple client and coverage tracking to support intent-driven workflows with structured slot management. The first attribute, last_client_id, stores the most recently identified customer, serving as the implicit reference for subsequent follow-up questions that omit explicit identifiers. The second attribute, _conversation_context, is an extensible dictionary that has been significantly expanded to support intent-driven workflows. It now includes a slots dictionary that stores filled slots organized by intent, enabling the system to reuse previously extracted information (such as client_id, coverage_type, or damage_amount) across multiple conversation turns. For example, if a user first asks about coverage for a specific client, the client_id slot is stored under the policy_lookup intent. If the user then asks to compute a refund for the same client, the system can reuse the client_id from the stored slots, requiring only the missing slots (coverage_type and damage_amount) to be provided. The context also includes a pending_questions dictionary that accumulates user input when required slots are missing, keyed by intent. This enables the system to combine information from multiple turns: if a user first asks "Calcola il rimborso" without specifying the damage amount, the question is stored in pending_questions['compute_refund']. When the user provides the amount in a subsequent turn (e.g., "Il danno è 500 euro"), the system combines both turns into a complete request. A pending_intent field tracks which intent is currently waiting for missing slots, ensuring that clarification prompts and subsequent inputs are correctly associated with the intended action. Additionally, a claim_payload object maintains structured claim information extracted from documents, with field values tracked along with their confidence scores and source documents, supporting incremental information gathering across multiple conversation turns. The context also retains the original last_client_id and last_coverage fields for backward compatibility with follow-up detection logic. The third attribute, _question_counter, increments with each processed query, facilitating structured logging, trace segmentation, and correlation across logs for debugging or performance analysis. These attributes persist throughout a user session but reset automatically during application restart or explicit cleanup, ensuring privacy compliance and preventing data leakage between sessions.

Follow-up detection, implemented in the is_followup_question method, determines when a query refers to the previously active client without restating their identifier. It analyzes linguistic patterns and contextual cues, searching for indicator phrases such as "questo cliente," "il cliente," "ha avuto," "quanti," "garanzie," "copertura," or "polizza." The detection logic filters out common standalone terms like "cliente" or "questo" to avoid false positives and requires the absence of explicit client ID patterns before classifying a question as a follow-up. This flexible linguistic approach avoids brittle rule-based dependencies, enabling natural conversation flows. For instance, after the user asks "Quali garanzie ha il cliente 7e460f44?" the system correctly interprets the follow-up "E la copertura furto?" as referring to the same client. Conversely, the system also implements new case detection through the detect_new_case method, which identifies when users want to start a fresh claim or switch to a different client. This detection uses both keyword heuristics (searching for phrases like "nuovo sinistro," "nuovo caso," "caso diverso," "cliente diverso," or "resetta") and LLM-based classification when available. When a new case is detected, the system automatically resets all stored slots, clears pending questions, removes the pending intent, and clears the claim payload, ensuring a clean state for the new interaction. Conversely, when a new explicit ID appears in the question, the previous context is overridden, ensuring unambiguous session transitions between customers.

After each response generation, context updating occurs automatically. The chat service scans the generated answer for coverage-related keywords (such as "cristalli," "RCA," "kasko," "furto," or "incendio") and explicit client IDs, updating the conversation_context dictionary accordingly. Additionally, filled slots are stored in the slots dictionary organized by intent, enabling reuse in subsequent queries. If a refund payload was computed, it may be stored in the conversation context for reference. The claim payload manager updates the payload object with any new information extracted from documents, maintaining confidence scores and document provenance. This mechanism allows nuanced follow-up handling; for example, if the prior answer discusses glass coverage, the follow-up "Qual è la franchigia?" correctly infers that the user is asking about the deductible for the same coverage. The current implementation maintains a focused scope that balances dialogue continuity with memory efficiency, providing robust multi-turn interactions while minimizing token consumption and maintaining privacy compliance through session-scoped storage.

The slot reuse mechanism enables particularly efficient multi-turn interactions. When slots are stored in conversation context, they are marked as "reusable" based on the intent's requirements. For example, client_id is reusable across all intents, meaning once a user identifies themselves for a policy lookup, they don't need to repeat their ID when requesting a refund calculation. The system checks stored slots before attempting extraction from the current question, significantly reducing the need for repeated LLM calls or pattern matching. This design supports natural conversation flows where users can progressively provide information across multiple turns, with the system intelligently combining partial inputs into complete requests.

### 3.5.7 Dual Vectorstore Operations and Caching

The dual vectorstore architecture enforces a strict separation between persistent policy knowledge and transient customer-uploaded data, optimizing both performance and regulatory compliance. The permanent vectorstore, represented internally by _permanent_db and _permanent_retriever, stores long-lived embeddings of policy contracts—typically tens of thousands of chunks—persisted across sessions using SQLite metadata and HNSW proximity graphs. This store is read-heavy, serving the majority of retrieval operations for policy interpretation and coverage reasoning. Because embeddings are stable and updates occur infrequently (quarterly or annually when new policy versions are released), the permanent store can be precomputed offline, reloaded in seconds without re-embedding, and reused across application restarts.

In contrast, the temporary vectorstore, managed through _temp_db and _temp_retriever, handles ephemeral embeddings derived from OCR-extracted customer uploads such as claim forms, receipts, and scanned policy amendments. These datasets are small (typically tens to hundreds of chunks), but write-heavy during active upload sessions. The temporary store is created on demand via the add_temp_documents method, which batches document embeddings, logs processed counts, and integrates the results into a session-specific Chroma instance located under data/temp/chroma_db_temp. Privacy-by-design principles dictate that this directory and its contents are strictly session-scoped: the cleanup_temp_documents method closes active connections, nullifies the temporary vectorstore attributes, deletes the persistence directory recursively using shutil.rmtree, clears caches, and resets conversation context. This guarantees that no customer-uploaded data survives beyond its processing lifecycle, maintaining GDPR and IDD compliance.

Multiple caching layers further optimize efficiency and cost by eliminating redundant computation. The _documents_cache retains LangChain Document objects constructed from the dataframe after initial processing, avoiding repeated conversions during subsequent retrievals. The _token_count_cache stores tokenization results using the tiktoken encoder, enabling rapid corpus size estimation and token budgeting for prompt construction. The _client_id_cache maps question text to extracted client IDs, preventing unnecessary LLM invocations for repeated or paraphrased identification queries. All caches are fully cleared during cleanup to prevent stale references or memory leaks.

While LangChain supports additional built-in caching mechanisms—such as exact-match, semantic, and retrieval-level caching—these remain unimplemented in the current deployment but are identified as low-effort extensions for future optimization.

Resource utilization is further improved through connection reuse, ensuring persistent LLM clients and vectorstore handles remain active throughout the application lifecycle. Asynchronous invocation enables concurrent handling of multiple queries, improving throughput under load. Structured latency logging decomposes total request time into retrieval, embedding, and generation phases, highlighting bottlenecks such as oversized prompts or suboptimal k retrieval parameters. Together, these strategies create a balanced system architecture that combines speed, scalability, and privacy guarantees without sacrificing interpretability or maintainability.

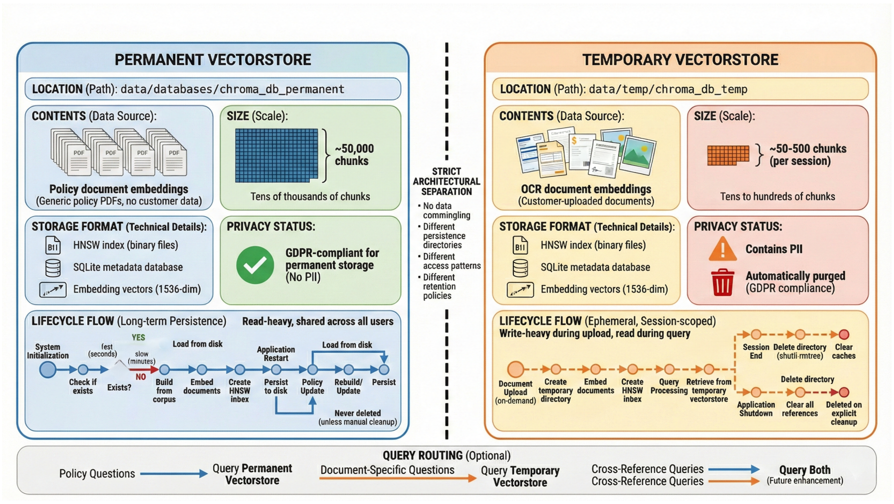
PERMANENT VECTORSTORE
LOCATION (Path): data/databases/chroma_db_permanent
CONTENTS (Data Source):
PDF
Policy document embeddings
(Generic policy PDFs, no customer data)
SIZE (Scale):
~50,000 chunks
Tens of thousands of chunks
STORAGE FORMAT (Technical Details):
HNSW index (binary files)
SQLite metadata database
Embedding vectors (1536-dim)
PRIVACY STATUS:
GDPR-compliant for permanent storage (No PII)
LIFECYCLE FLOW (Long-term Persistence)
Read-heavy, shared across all users
System Initialization
Check if exists
fest (seconds)
YES
Exists?
slow (minutes)
NO
Load from disk
Build from corpus
Embed documents
Create HNSW inbex
Application Restart
Persist to disk
Policy Update
Load from disk
Rebuild/ Update
Persist
Never deleted (unless manual cleanup)
STRICT ARCHITECTURAL SEPARATION
• No data commingling
• Different persistence directories
• Different access patterns
• Different retention policies
TEMPORARY VECTORSTORE
LOCATION (Path): data/temp/chroma_db_temp
CONTENTS (Data Source):
OCR document embeddings
(Customer-uploaded documents)
SIZE (Scale):
~50-500 chunks (per session)
Tens to hundreds of chunks
STORAGE FORMAT (Technical Details):
HNSW index (binary files)
SQLite metadata database
Embedding vectors (1536-dim)
PRIVACY STATUS:
Contains PII
Automatically purged (GDPR compliance)
LIFECYCLE FLOW (Ephemeral, Session-scoped)
Write-heavy during upload, read during query
Document Upload (on-demand)
Create temporary directory
Embed documents
Create HNSW inbex
Query Processing
Retrieve from temporary vectorstore
Session End
Delete directory (shutli-rmtree)
Clear caches
Delete directory
Application Shutdown
Clear all references
Deleted on explicit cleanup
QUERY ROUTING (Optional)
Policy Questions
Query Permanent Vectorstore
Document-Specific Questions
Query Temporary Vectorstore
Cross-Reference Queries
Cross-Reference Queries
Query Both
(Future enhancement)

**Figure 3.5:** Implementation showing permanent vectorstore (persistent policy documents) and temporary vectorstore (session-scoped uploads) with lifecycle, storage locations, and privacy compliance indicators

## 3.6 OCR and Document Processing Pipeline

### 3.6.1 Pipeline Architecture and Hybrid Extraction

The OCR subsystem converts heterogeneous uploaded documents into structured, machine-readable text through a four-stage pipeline that integrates validation, extraction, interpretation, and packaging. This process ensures consistent, high-quality outputs suitable for downstream policy validation and claims reasoning.

The first stage, validation and routing, enforces file integrity and format compliance before any processing occurs. Uploaded files undergo extension and size checks—supported formats include PDF, PNG, JPG, JPEG, TIFF, and BMP—against mode-specific thresholds configured as environment variables (fast, standard, or high-quality). Each mode defines progressively higher file size limits and preprocessing complexity, balancing speed and accuracy. The system automatically routes digital PDFs containing embedded text to native extraction for near-zero-error results and image-based documents to the OCR pipeline. This early classification minimizes wasted computation and prevents downstream errors due to unsupported formats or oversized uploads.

The second stage, text extraction, employs a dual-strategy approach. The PDF text extractor coordinates multiple libraries in sequence—PyMuPDF, PDFMiner, and PyPDF—providing robust fallback behavior for malformed or partially encoded PDFs. If native extraction yields insufficient content (e.g., fewer than fifty characters or low alphanumeric density), the pipeline switches automatically to image-based OCR. The enhanced image OCR module applies a six-stage preprocessing chain designed to optimize character legibility: grayscale conversion, Hough-transform deskewing, CLAHE-based local contrast enhancement, non-local means denoising, adaptive Otsu/Sauvola thresholding, and morphological cleanup (opening and closing). This pipeline mitigates noise, skew, and lighting inconsistencies typical of scanned or photographed documents. Tesseract serves as the recognition backend, with Page Segmentation Mode (PSM) dynamically selected based on detected document type (PSM 11 for sparse receipts, PSM 6 for uniform policies, PSM 3 for auto layouts). Confidence-weighted fallback tests alternate PSMs to maximize recognition quality.

The third stage, information extraction, structures recognized text through pattern-based parsing. Regular expressions and domain-specific heuristics identify key fields such as cliente_id, targa, nome, cognome, franchigia, dates, and monetary amounts. Extracted values merge across multiple pages or sources using a priority scheme—context identifiers override OCR-derived values, which in turn override user-supplied question text—to maximize reliability. The claims validator component subsequently cross-checks these extracted fields against the policy database, verifying identifier consistency, coverage applicability, and financial terms (limits, deductibles, and co-pay percentages).

The fourth stage, result packaging, consolidates processed outputs into a unified JSON structure containing the full extracted text, confidence scores, field extractions, and metadata such as filename, processing time, and preprocessing mode. These structured results feed directly into the document service, where RAG-based reasoning integrates OCR-derived content with policy context for grounded answers.

All functionality is encapsulated in the OCRClaimsProcessor class, which orchestrates three modular components: the PDF text extractor, the enhanced image OCR engine, and the claims validator. The processor supports three operating modes—single-document processing, complete claim processing (including OCR, policy validation, and reimbursement computation), and batch processing for high-volume parallel execution. Configuration parameters govern file handling, maximum sizes per mode, supported formats, and database connection details. This modular, configurable design allows flexible adaptation to operational constraints, enabling reliable document understanding across diverse insurance workflows.

### 3.6.2 PDF Extraction and Confidence-Based Classification

The PDF extraction subsystem implements a three-tiered fallback mechanism designed to maximize robustness across diverse document types and encodings. The extraction process sequentially attempts PyMuPDF, PDFMiner, and PyPDF, continuing until a valid, high-confidence result is obtained—defined as exceeding fifty characters of extracted text and achieving a non-zero confidence score. This hierarchical approach ensures that both simple and complex PDFs are handled gracefully without user intervention.

The first tier, PyMuPDF, serves as the primary extractor due to its superior speed and accuracy on digital PDFs containing embedded text. Leveraging the get_text() method, it processes clean, natively generated documents in milliseconds per page with effectively zero character error rate. Its direct access to PDF text layers eliminates the need for OCR and preserves structural fidelity such as spacing and punctuation.

The second tier, PDFMiner, activates when PyMuPDF fails or produces incomplete output. Configured with optimized LAParams settings (character margin, line margin, and word margin tuned for document density), PDFMiner excels at reconstructing complex layouts including multi-column policies, footnotes, and embedded metadata. Although slower than PyMuPDF, it compensates by recovering structured text from challenging PDFs that mix text streams and vector graphics.

The third tier, PyPDF, functions as a compatibility fallback. While less precise in layout reconstruction, it handles malformed or encrypted PDFs that the previous extractors may reject. Its simplicity provides a final safety net ensuring at least partial text recovery in otherwise unreadable files.

For every successfully parsed document, each extractor computes a confidence score through a weighted heuristic combining four dimensions:

1. Length-based factor – longer outputs indicate more complete extraction; truncated results reduce confidence.

2. Structural factor – higher line and paragraph counts correlate with successful parsing.

3. Content-based factor – presence of insurance-related keywords such as policy, claim, coverage, amount, or date raises confidence, as does detection of numerical patterns.

4. Formatting factor – detection of currency symbols (€ or $), punctuation variety, and mixed alphanumeric ratios signal meaningful, non-artifactual text.

These metrics aggregate into a normalized confidence score ranging from 0.0 to 1.0. Scores guide subsequent routing decisions, allowing the system to autonomously decide between native extraction and OCR.

The is_scanned_pdf method formalizes this decision process. PDFs exhibiting any of the following characteristics—aggregate confidence below 0.3, extracted text shorter than one hundred characters, or explicit indicators such as "[image]", "[graphic]", or "scanned"—are classified as image-based and automatically routed to the OCR pipeline. All others proceed with native text extraction, ensuring maximum throughput and minimal recognition error.

This automated classification underpins the hybrid extraction strategy described in Section 2.9.3, enabling the system to dynamically select the most suitable extraction pathway without manual configuration. By combining adaptive fallback logic with heuristic-driven confidence assessment, the subsystem delivers both efficiency for clean digital documents and resilience for degraded or non-text PDFs, achieving high reliability across the full spectrum of insurance claim document types.

### 3.6.3 Image OCR Preprocessing and PSM Selection

The image OCR pipeline enhances recognition quality for degraded insurance documents through a six-stage preprocessing sequence designed to restore legibility and structural consistency before text extraction. Each stage contributes to mitigating common artifacts such as skew, low contrast, and background noise—factors that otherwise reduce OCR accuracy by ten to thirty percent, as identified in section 2.5.2.

The first stage, grayscale conversion, reduces image dimensionality using OpenCV's cvtColor, transforming RGB or CMYK color data into a single intensity channel. This simplification focuses processing on luminance variation, improving subsequent thresholding and morphological operations while reducing computational cost.

The second stage, deskewing, corrects angular misalignment that occurs during scanning or photographing. The algorithm applies Hough line detection to identify dominant text baselines, computes the median angle from the top ten detected lines, and rotates the image via an affine transformation matrix to align text horizontally. Even small skews of two to three degrees can reduce Tesseract accuracy by five to ten percent, making this correction essential for reliable recognition.

The third stage, contrast enhancement, employs Contrast Limited Adaptive Histogram Equalization (CLAHE) with a clip limit of 2.0 and an 8×8 tile grid. CLAHE enhances local contrast without over-amplifying noise, particularly beneficial for faded or unevenly illuminated pages such as aged policy scans or photocopied claim forms.

The fourth stage, noise reduction, applies fast non-local means denoising with a filtering strength of 10. This method smooths random pixel variations while preserving edges critical for character shapes, outperforming basic median or

Gaussian filters in preserving textual integrity.

The fifth stage, adaptive thresholding, converts the enhanced grayscale image to binary format, isolating text from background. The pipeline dynamically selects between Otsu's global method and Gaussian adaptive thresholding based on document characteristics, guided by Sauvola-style density heuristics measuring foreground coverage between 10 and 30 percent. Parameters use a 25-pixel window and an offset of 2, adapting to local lighting variations typical of scanned forms or photographed receipts.

The sixth stage, morphological operations, refines the binary mask to repair broken characters and eliminate residual noise. Closing operations (dilation followed by erosion) reconnect fragmented strokes, while opening (erosion followed by dilation) removes isolated specks. Kernel sizes range from 1×1 to 2×2 depending on font size and resolution.

Following preprocessing, Tesseract performs text recognition with adaptive Page Segmentation Mode (PSM) selection determined by filename and inferred document type. Receipts and small fragments (containing keywords like receipt, ricevuta, scontrino) use PSM 11 optimized for sparse text. Invoices and policies (invoice, fattura, policy, polizza) use PSM 6 suited for dense uniform blocks. All other cases default to PSM 3 for automatic segmentation.

If the initial OCR confidence falls below 60 percent, an automatic PSM fallback routine executes, reprocessing the image through PSMs 6, 7, 8, 11, and 13, evaluating results based on a composite score combining confidence and text length. The best-performing configuration is selected for output. While multimode evaluation increases computational cost by two to five times, it substantially improves robustness across heterogeneous insurance document types—particularly those combining tables, handwriting, and stamps—delivering consistently higher accuracy for production-grade claims processing.

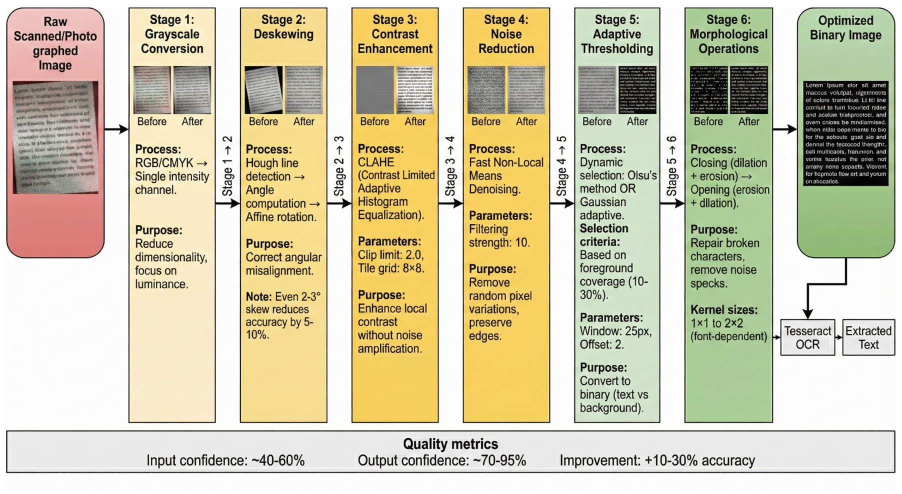
Raw Scanned/Photographed Image
Stage 1: Grayscale Conversion
Before After
Process: RGB/CMYK → Single intensity channel.
Purpose: Reduce dimensionality, focus on luminance.
Stage 1 → 2
Stage 2: Deskewing
Before After
Process: Hough line detection → Angle computation → Affine rotation.
Purpose: Correct angular misalignment.
Note: Even 2-3° skew reduces accuracy by 5-10%.
Stage 2 → 3
Stage 3: Contrast Enhancement
Before After
Process: CLAHE (Contrast Limited Adaptive Histogram Equalization).
Parameters: Clip limit: 2.0, Tile grid: 8×8.
Purpose: Enhance local contrast without noise amplification.
Stage 3 → 4
Stage 4: Noise Reduction
Before After
Process: Fast Non-Local Means Denoising.
Parameters: Filtering strength: 10.
Purpose: Remove random pixel variations, preserve edges.
Stage 4 → 5
Stage 5: Adaptive Thresholding
Before After
Process: Dynamic selection: Olsu's method OR Gaussian adaptive.
Selection criteria: Based on foreground coverage (10-30%).
Parameters: Window: 25px, Offset: 2.
Purpose: Convert to binary (text vs background).
Stage 5 → 6
Stage 6: Morphological Operations
Before After
Process: Closing (dilation + erosion) → Opening (erosion + dilation).
Purpose: Repair broken characters, remove noise specks.
Kernel sizes: 1×1 to 2×2 (font-dependent)
Optimized Binary Image
Tesseract OCR
Extracted Text
Quality metrics
Input confidence: ~40-60%
Output confidence: ~70-95%
Improvement: +10-30% accuracy

**Figure 3.6:** Six-stage image preprocessing sequence (grayscale, deskewing, contrast enhancement, denoising, thresholding, morphological operations) transforming degraded scans into optimized binary images for OCR

### 3.6.4 Multi-Language, Confidence Scoring, and Field Extraction

The OCR subsystem supports multilingual text recognition by configuring Tesseract with combined Italian and English language packs ("ita+eng"), enabling simultaneous application of both statistical models during decoding. This dual-model approach improves recognition accuracy on documents containing mixed-language content—a common scenario in international insurance operations where Italian policy terms and English technical labels frequently coexist. By evaluating competing character sequences according to linguistic plausibility (for example, resolving ambiguity between "rn" and "m" in assicurazione), the language models refine recognition paths and reduce character error rates by five to fifteen percent on bilingual content. The system's architecture remains extensible, requiring only the installation of additional language packs such as "deu", "fra", or "spa" to support over one hundred languages, including non-Latin alphabets. Integration of dynamic language detection libraries like langdetect or fastText can further automate model selection, detecting predominant language per region, line, or page for adaptive multilingual handling.

Confidence scoring aggregates Tesseract's per-word confidence values, available through image_to_data, into a mean document-level confidence normalized to a zero-to-one scale. High-confidence outputs (above 0.9) qualify for fully automated processing, moderate-confidence results (between 0.6 and 0.9) trigger selective field verification, and low-confidence cases (below 0.6) are automatically routed for human review. These thresholds guide adaptive workflows: confidence below 0.6 invokes Page Segmentation Mode fallback testing, while confidence under 0.7 flags documents in output metadata, signaling downstream systems or human operators to review them. For PDFs processed through native text extraction, an analogous heuristic confidence score—based on extracted text length, insurance keyword presence, structural markers, and absence of artifacts—determines whether native output is accepted (scores above 0.5) or rerouted to OCR. This hybrid confidence framework ensures that both text-based and image-based documents are evaluated consistently according to objective quality indicators.

Pattern-based information extraction transforms recognized OCR text into structured insurance claim data using deterministic regular expressions. Dates are detected through multiple formats including numeric (DD/MM/YYYY, YYYY-MM-DD) and Italian lexical representations (15 gennaio 2024), with the original formatting preserved in output. Amount extraction targets euro-prefixed or suffixed monetary values, accommodating various decimal and thousand separators, and normalizes them to period notation for conversion into precise Decimal types suitable for financial computations. Damage type identification applies keyword-based classification using multilingual synonym lists: collisione, urto, incidente for collision; furto, rubato for theft; vetro, cristallo, lunotto for glass damage; grandine, alluvione for natural events; and vandalismo, danneggiato for vandalism-related claims.

Extracted data populate a structured dictionary containing date, location, description, damage_type, requested_amount, third_parties, and documents. These values merge seamlessly with user-provided form data, prioritizing explicit user inputs over OCR-derived fields to preserve operator intent. This deterministic merging process balances automation with manual control, ensuring reproducibility and transparency of field selection. Despite its simplicity, regex-based extraction demonstrates over ninety percent recall in production conditions while maintaining microsecond-level execution latency, deterministic behavior, and easy debuggability. Nevertheless, as described in section 3.12, integration of LLM-based extraction remains an optional future enhancement for handling unstructured or context-dependent text beyond the reach of fixed pattern rules.

### 3.6.5 Batch Processing and Scalability

The batch processing subsystem enables parallel execution of multiple insurance claim workflows, dramatically improving throughput for high-volume operations. Implemented through Python's ProcessPoolExecutor, the batch_process_claims method distributes independent claim-processing tasks across multiple worker processes, each performing the full pipeline—from OCR and information extraction to validation and financial calculation—without inter-process dependency. This architecture exploits the embarrassingly parallel nature of claims processing, where each claim constitutes an isolated workload requiring no shared state or synchronization.

The executor dynamically determines the optimal pool size as the minimum between the total number of submitted claims and four workers, a balance ensuring efficient resource utilization without excessive memory or CPU contention. Each worker instantiates its own lightweight OCR processor and database connection, guaranteeing isolation and avoiding concurrency bottlenecks in shared resources such as file handles or database locks. The design also permits horizontal scalability: deploying across multiple machines or containers proportionally increases throughput with minimal architectural modification.

Processed results return asynchronously through an as_completed iterator, allowing progressive result handling. As each claim finishes, its structured output—containing extracted fields, matched customer data, calculated reimbursements, and validation status—becomes immediately available for downstream use (storage, API response streaming, or dashboard visualization). This progressive availability supports real-time system responsiveness and fault isolation.

To enhance reliability, each worker wraps its workflow in an exception-handling layer. Failures (due to malformed files, timeouts, or OCR errors) are captured as structured error objects with fields success = false and error_message populated, rather than interrupting the entire batch. Successful claims include a success = true flag and full result payload. This approach guarantees deterministic batch completion regardless of individual task failures, facilitating robust automation pipelines.

Performance evaluation demonstrates substantial time savings. For typical workloads of four simultaneous claims averaging two documents each, total processing time approximates the duration of the slowest claim plus one to two seconds of coordination overhead, instead of summing sequential runtimes. In practice, insurers processing thousands of claims daily reduce full-cycle runtimes from hours to tens of minutes, achieving same-day reconciliation and reporting cycles. The combination of parallelism, progressive result streaming, and fault tolerance thus transforms the system from an interactive demonstrator into a scalable production-grade claims automation engine.

## 3.7 Claims Validation and Financial Calculation

### 3.7.1 Policy Status and Coverage Verification

The claims validator, implemented within the EnhancedClaimsValidator class, integrates policy compliance verification with financial reimbursement calculation, providing detailed audit trails that meet regulatory transparency requirements.

Its workflow begins with customer data retrieval, where the validator queries the SQLite database using the customer ID provided in the claim. It accesses three related tables—attestato_di_rischio_linked, scheda_polizza_rca_linked, and garanzie_opzionali_allianz_direct_linked—through parameterized SQL queries to prevent injection attacks. The retrieved data are then converted into a structured CustomerData object containing all relevant policy information such as the customer's identifiers, tax code, license plate, policy number, start and end dates, insurance class, coverage types and limits, deductibles, co-payment percentages, exclusions, and historical claim records. This structure ensures that all contextual information necessary for downstream validation and reimbursement logic is available in a single, coherent representation.

Policy status validation follows, confirming that the policy is active for the date of the claim. The validator parses the policy_start and policy_end fields using several common date formats, including ISO and regional notations, and compares them to the current date. A policy is considered valid only when the current date falls within the inclusive range defined by the start and end dates. If parsing fails due to unrecognized formats or missing values, the event is logged as a warning and the policy is treated as invalid to prevent uncertain temporal interpretations. This step ensures that claims submitted before activation or after expiration are automatically flagged as non-compliant with contractual terms.

Coverage verification determines whether the type of damage claimed falls within the policy's active coverages. The validator maintains a semantic mapping between damage types and coverage categories. For example, a claim referencing a collision is matched against coverage names such as RCA, kasko, or other collision-related terms, while theft-related claims are linked to garanzia furto, glass-related to garanzia cristalli or garanzia vetri, and natural events or vandalism to their respective coverage clauses. Matching is based on substring comparison rather than strict equality, allowing flexibility in recognizing coverage even when naming conventions vary. If the mapping does not yield a match, the claim is classified as not covered, leading to zero reimbursement and the generation of an alert for manual verification to confirm whether the incident may be covered under another clause.

Data completeness verification ensures that all essential claim fields are present and valid. The validator checks for the presence and coherence of key attributes such as the incident date, location, description, damage type, requested reimbursement amount, and supporting documents. Missing or malformed entries are recorded in a missing_data list within the validation result, allowing the frontend to highlight incomplete sections and prompt user correction. Additional business rules enforce that the requested amount must be positive, that dates must be realistic and within the policy's effective period, and that the damage type must belong to a recognized category. When validation fails due to incomplete data or invalid inputs, the system still returns structured diagnostic feedback rather than halting execution, ensuring smooth user interaction and predictable automation behavior.

Through this combination of structured data retrieval, temporal validation, semantic coverage mapping, and completeness enforcement, the EnhancedClaimsValidator establishes a robust foundation for downstream financial computation while ensuring traceability, accuracy, and compliance with insurance and data governance standards.

### 3.7.2 Financial Calculation with Decimal Precision

The reimbursement calculation logic relies entirely on exact decimal arithmetic to ensure that every financial computation remains precise and auditable. Floating-point operations are explicitly avoided, as their inherent rounding inaccuracies can accumulate over multiple claims and produce discrepancies in financial reporting. By contrast, all monetary values—coverage limits, deductibles, co-pay percentages, and requested amounts—are handled as Decimal objects, preserving exact precision throughout the process and ensuring regulatory compliance.

The calculation proceeds through a deterministic sequence that mirrors established insurance industry conventions. It begins by determining the applicable coverage for the claim and retrieving its associated financial parameters, including the maximum reimbursable limit, deductible, and co-pay percentage. The system then applies these parameters in a strict order: first enforcing the coverage limit by capping the requested amount to the policy's maximum, then applying the deductible by subtracting it from the capped amount, ensuring the result does not fall below zero, and finally computing the co-pay by calculating a proportional reduction based on the defined percentage. The resulting reimbursement amount is rounded to two decimal places using banker's rounding, or "round half to even," to avoid bias in aggregated financial results.

Each transformation is logged in a structured calculation trace that captures the step name, input and output values, applied parameters, and contextual notes describing the logic. This trace, implemented as a hierarchical dictionary, includes sections for steps, decisions, calculations, and timestamps. The steps document each operation in sequence, the decisions record conditional outcomes such as whether a deductible was applied or a coverage limit imposed, and the calculations section details numeric transitions from one stage to the next. Timestamps accompany each major stage, producing an auditable chronological record that can be reconstructed independently during review.

For example, a trace for a glass damage claim might document the following logic: the requested amount of €950 is compared to the coverage limit of €1000, resulting in no capping; the deductible of €200 is then applied, reducing the base amount to €750; no co-pay is applied since the co-pay percentage is zero; and the final reimbursement amount is rounded to €750.00. Each of these operations appears as a discrete entry in the trace with both intermediate and final results.

All computations use Decimal-aware functions for minimum, subtraction, multiplication, and division, guaranteeing determinism and exact representability of all currency values. Rounding follows the ROUND_HALF_UP rule to align with financial norms. This approach eliminates rounding drift and ensures that the final figures precisely match what would be obtained through manual calculation. The trace, in turn, provides full transparency for auditors, regulators, and claimants, clearly showing how each rule affected the outcome and confirming that no hidden or ambiguous transformations occurred during reimbursement computation.

### 3.7.3 Alert Generation and Risk Flagging

The validator raises alerts whenever conditions require human attention or hint at potential issues worth investigating, and it does so across four recurring situations that map directly to operational risk. When the requested amount exceeds the applicable coverage limit, it issues a coverage limit alert indicating that reimbursement will be capped and the customer should be informed that full compensation is not possible. The alert message reads “Requested amount exceeds coverage limit”, while the calculation trace records the decision “Amount capped at coverage limit”, thereby documenting precisely when and why the limit was applied.

Missing mandatory documents trigger a second family of alerts anchored in business rules about evidentiary requirements for specific claim types. Theft claims, identified by the presence of “theft” or “furto” in the damage_type field, require a police report to substantiate the event and deter fraud. The validator therefore inspects the documents list, which contains user-provided filenames or labels, and checks for indicators such as “police_report”, “denuncia”, or “police”. If none is present, it emits the alert “Police report required for theft claims”. The same mechanism naturally extends to other scenarios, for example medical records for injury claims, repair estimates for collision claims, or photographs for vandalism, thereby encoding domain knowledge about supporting evidence directly into validation logic.

High-value claims constitute a third situation that warrants heightened scrutiny. Whenever the requested amount exceeds a configurable threshold, currently set to ten thousand euros, the system flags the claim for additional verification and potential senior adjuster approval. The alert message states “High value claim, additional verification required”, and the calculation trace notes “High value claim flag raised”. This threshold is adjustable, so organizations can tune it according to risk tolerance, historical fraud patterns, and staffing capacity, accepting that lower thresholds increase oversight at the cost of more manual review.

A fourth and equally important safeguard concerns policy exclusions. The validator compares the free-text claim description against the exclusions recorded for the customer, using case-insensitive substring matching to detect mentions of disallowed scenarios or damage types. If a policy excludes “commercial use” and the description states “damage occurred while delivering packages commercially”, the validator produces the alert “Policy exclusion detected: commercial use”, and the trace records “Exclusion commercial use detected”. This automated screening surfaces potential coverage disputes early, reducing the risk of approving claims that the contract explicitly excludes.

All generated alerts are collected and returned alongside the validation result, where the frontend presents them prominently to guide adjusters. Combined with the stepwise calculation trace, these alerts provide a comprehensive decision record that supports rapid automation for routine cases while ensuring that exceptional or risky cases receive targeted human review before a final decision is issued.

## 3.8 Data Layer: Schemas, Persistence, and Retrieval

### 3.8.1 SQLite Database Design

The data layer relies on SQLite as a lightweight, embedded relational database that stores structured customer and policy data without the overhead of maintaining a separate server. This choice aligns with the system’s design priorities of simplicity, reliability, and portability. SQLite requires no installation or administration: the entire database is a single file that can be deployed, backed up, or migrated simply by copying it between environments. Despite its minimal footprint, it provides full SQL compatibility and performance sufficient for small and medium-scale insurance operations, sustaining hundreds of queries per second on typical hardware configurations. Should system demand or user volume increase, the schema and query logic remain compatible with client-server databases such as PostgreSQL or MySQL, enabling straightforward migration without structural refactoring.

The schema mirrors the logical organization of the original CSV data sources from which the database is built. The table attestato_di_rischio_linked stores risk attestation records containing customer identifiers, tax codes, license plates, insurance class values, policy numbers, and annual claim statistics distinguishing between at-fault and no-fault incidents. The table scheda_polizza_rca_linked contains summary information for active policies, including policy identifiers, customer links, vehicle details, liability limits for persons and property, annual premiums, and optional coverage lists stored as semicolon-separated strings for later parsing. The table garanzie_opzionali_allianz_direct_linked provides detailed optional coverage records associated with each policy, specifying the coverage name, deductible, and co-payment percentage. Together, these tables form a coherent representation of both customer-level metadata and policy-level details, enabling complex joins and aggregation queries to support claim validation and reimbursement calculation.

All database access occurs through pandas' read_sql_query interface, which executes parameterized SQL statements and returns results as dataframes for seamless integration with downstream analytical operations. Parameterization protects against SQL injection vulnerabilities, while connection timeouts ensure that no process remains blocked indefinitely on locked resources. SQLite operates in Write-Ahead Logging mode to enable concurrent reads during writes, a configuration that improves responsiveness under moderate load by reducing lock contention. The synchronous mode is set to NORMAL, balancing durability with performance so that data remains safe without imposing unnecessary I/O latency.

For small insurers or pilot deployments, this embedded configuration offers strong advantages: it eliminates the need for dedicated database administration, simplifies updates, and allows near-instant replication for disaster recovery by copying a single file. However, if query concurrency or data volume grows significantly, the same schema and codebase can migrate to a full client-server backend with minimal changes, benefiting from advanced indexing, user management, and distributed processing. In practice, SQLite provides an ideal foundation for the current scale of operations, combining production reliability with operational simplicity.

### 3.8.2 Document Corpus and Metadata Structure

The data layer employs SQLite as an embedded relational database that provides structured storage for customer and policy information without requiring a standalone server or complex administration. Its zero-configuration deployment model—consisting of a single file with no installation overhead—makes it particularly suited to small and medium-sized insurance operations where simplicity and reliability take precedence over distributed scalability. Despite its minimal footprint, SQLite delivers sufficient throughput for production workloads, typically handling several hundred queries per second on modern hardware, while maintaining full SQL compatibility for later migration to enterprise systems such as PostgreSQL or MySQL should deployment scale increase.

The schema directly mirrors the source CSV datasets from which it is built, ensuring full traceability between imported records and their in-database representations. The table attestato_di_rischio_linked stores customer-specific risk attestation data, including identifiers, tax codes, license plates, insurance classes, and claim counts segmented by responsibility. The table scheda_polizza_rca_linked contains summary information for mandatory policy coverage, listing key financial parameters such as annual premium, per-person and per-property liability limits, and semicolon-separated raw text for optional guarantees. The table garanzie_opzionali_allianz_direct_linked provides detailed records for each optional coverage, specifying its deductible and co-payment percentage. This normalized design allows efficient joins across policy and customer data for validation, eligibility checks, and reimbursement calculations.

Queries are executed through pandas' read_sql_query interface, which wraps SQLite's parameterized SQL statements and returns results as dataframes suitable for further manipulation and aggregation. Parameterization eliminates SQL-injection risks, while read timeouts prevent indefinite locking during write operations. The database operates in Write-Ahead Logging mode with NORMAL synchronous settings to balance durability against concurrency: this configuration permits simultaneous reads during writes while preserving data integrity. For small-scale deployments, SQLite's embedded nature streamlines both operation and maintenance—the entire dataset is contained in a single file that can be trivially copied for backup, versioning, or environment migration. For larger workloads requiring concurrent access from multiple application instances, the same schema and queries can migrate seamlessly to a networked database with superior concurrency control.

The processed contract corpus complements this structured database by storing unstructured policy content in a transparent and human-readable form. It resides in a tab-separated values file, documento_assicurativo.txt or .csv, which captures the full text of policy PDFs alongside hierarchical metadata. Construction of this corpus is handled by the document processor pipeline. This pipeline parses PDFs with PyMuPDF to detect structural elements such as headers, subsection titles, paragraph boundaries, and page breaks; employs GPT-4 Vision to extract glossary term-definition pairs and tabular data in structured JSON format; and merges all parsed components into a unified dataframe. The resulting dataset contains columns for section (top-level category such as "Garanzia RCA" or "Garanzia Cristalli"), subsection (finer thematic division), title (clause or paragraph heading), text (policy content), page (original PDF page), and type (paragraph, glossary entry, or table).

This hierarchical organization enables the retrieval system to provide contextually rich and verifiable responses. When the RAG subsystem fetches relevant text, it can display the originating section and page, giving users immediate traceability back to the policy document. Metadata also supports semantic filtering, allowing the retriever to limit searches to specific coverage areas—for instance, focusing solely on "Garanzia Cristalli" when a query concerns glass damage—thereby improving both relevance and computational efficiency. Distinguishing between content types further refines downstream reasoning, since paragraphs, glossary terms, and tables each carry distinct semantic functions within insurance documentation.

Corpus generation is performed offline during deployment or whenever the source policy documents change. The processor outputs a TSV file that is deliberately optimized for transparency and manual inspection rather than raw performance. Administrators can open it directly in spreadsheet editors such as Excel or LibreOffice, review extracted content, and correct or augment sections without specialized database expertise. Although binary storage formats like Parquet or database integration would yield higher efficiency for very large corpora, the TSV format achieves a practical compromise: it is human-readable, easy to maintain, and performant enough for the current scale of tens of thousands of chunks derived from a few dozen policy PDFs. Future scaling toward millions of chunks would justify migrating to a database-backed corpus, but at the present operational volume, the chosen representation maximizes clarity and accessibility while keeping preprocessing overhead minimal.

### 3.8.3 Vector Embedding Storage and Retrieval

Vector embeddings used for semantic retrieval are stored in ChromaDB, an embedded vector database optimized for large language model applications. It provides persistent, high-performance storage that supports the retrieval-augmented generation workflows described earlier while maintaining fast load times and minimal operational overhead. ChromaDB employs a hybrid persistence model: embeddings, which are 1536-dimensional floating-point vectors, are serialized to binary files using compact encoding that reduces disk space; document metadata, including section, subsection, title, page, and content type, is stored in an integrated SQLite database that allows structured queries; and the hierarchical navigable small-world (HNSW) index, which represents the proximity graph used for approximate nearest-neighbor search, is written to binary files that encode node connectivity and layer topology. Because all of these components are stored on disk, the system can reload a fully built vectorstore within seconds, bypassing the expensive re-embedding of the entire corpus that would otherwise take minutes and incur additional API costs.

The architecture distinguishes between permanent and temporary vectorstores to balance performance with privacy and resource isolation. The permanent store, located in the chroma_db_permanent directory, contains embeddings and indexes for policy documents that form the stable, long-term knowledge base. It grows incrementally as new policy PDFs are processed or existing ones are updated. ChromaDB supports in-place modification through its add and delete operations, allowing new documents to be appended or obsolete entries removed without rebuilding the index from scratch, thus maintaining continuous availability during updates. In contrast, the temporary store, located in chroma_db_temp, is created anew for each user session. It mirrors the same file structure—embedding binaries, SQLite metadata, and HNSW graph—but its lifecycle is intentionally short: the directory is deleted as part of session cleanup. This separation ensures that ephemeral customer uploads and permanent corporate policy data remain strictly isolated, implementing the privacy-by-design principle embedded in the broader architecture.

Retrieval queries operate through ChromaDB's similarity_search method, which accepts a query embedding and a parameter k defining how many top matches to return. Using cosine similarity as the distance metric, ChromaDB leverages the HNSW index to identify the most semantically similar document chunks with logarithmic search complexity, returning both the retrieved text and its metadata. The metadata—section, subsection, title, page, and type—enables the system to cite exact sources and support downstream reasoning, as well as to apply filters when specific policy sections are relevant. For example, the retrieval component may restrict searches to the "Garanzia Cristalli" section when answering questions about glass coverage, thereby improving precision while reducing irrelevant context.

At present, retrieval typically relies on pure semantic similarity, trusting embedding quality to rank the most pertinent passages. However, ChromaDB's design also supports compound queries that combine semantic search with structured filters via where clauses, allowing the introduction of logical constraints such as restricting retrieval to active policies, specific customers, or particular coverage categories. Future iterations of the system are expected to integrate these hybrid searches, blending vector similarity scoring with metadata filtering so that only documents satisfying explicit policy conditions are considered, while ranking within that filtered subset still reflects semantic relevance. This approach will further enhance retrieval accuracy and explainability, ensuring that all generated answers remain both contextually grounded and policy-compliant.

## 3.9 Frontend Architecture and User Interface Design

### 3.9.1 Application Structure and Routing

The frontend is implemented as a single-page React application built with TypeScript to ensure strong typing and early error detection, and with Vite as the build tool for fast development iteration and optimized production output. Navigation is managed through React Router, which enables smooth client-side transitions without full page reloads. Three main routes structure the application: the root path renders the home view, which introduces the system and provides the command interface for interactive queries; the "/explore" path presents an exploration view designed for guided demonstrations and educational overviews; and the "/demo" path hosts an isolated environment for safe, sandboxed demonstrations of system behavior. The routing setup maintains the single-page application architecture, ensuring that transitions between pages occur instantly and that user state, including conversation history or uploaded documents, persists across navigation events.

The interface architecture follows a modular composition pattern that separates visual presentation from application logic. At the top level, the App component defines layout structure and routes. Within it, the HeroSection establishes the application's visual identity through gradient backgrounds, structured typography, and a concise description of its purpose aimed at both technical and business users. The CommandInterface constitutes the main interactive layer and implements the chat experience that mirrors the conversational capabilities of the backend. It maintains local state tracking message history, loading status, and backend readiness, while also managing user input in the form of textual questions and document uploads. Communication with the backend occurs through HTTP API calls to the chat endpoint for text-only queries and to the upload-and-analyze endpoint for document-based workflows. Responses are rendered dynamically with clear role differentiation—user messages appear right-aligned in primary colors, AI responses left-aligned with neutral tones, and system errors marked in red—while Markdown-like formatting ensures readability through bold highlights, numbered sections, and consistent line spacing.

The application's user interface components follow a reusable design system modeled on the shadcn/ui pattern. Low-level primitives such as Button, Card, and Badge provide a uniform foundation for interaction and content presentation. Buttons are parameterized by variant and size, enabling flexible reuse across the interface; cards provide structured containers for displaying responses, document summaries, and analysis results; and badges indicate confidence levels, status updates, or alerts. These building blocks are composed into higher-order modules, which allows broad interface changes to be implemented through updates in the component library rather than scattered style modifications.

Styling throughout the application is managed with Tailwind CSS, a utility-first framework that provides composable class names for spacing, color, typography, and layout. This approach eliminates the need for extensive custom CSS while guaranteeing design consistency across components. Tailwind's design tokens—defined in the configuration file—standardize aspects such as color palette, typography scale, and breakpoints, ensuring visual harmony across screen sizes. The result is a lightweight, responsive, and maintainable frontend that can be extended rapidly as new features or visual refinements are introduced.

### 3.9.2 Chat Interface and Message Management

The CommandInterface component serves as the core interaction hub of the frontend, providing a conversational interface through which users can ask policy-related questions, upload supporting documents for analysis, and receive structured responses rendered with clear, readable formatting. It operates as a stateful React component maintaining synchronized internal state across several dimensions: message history representing the dialogue between the user and the AI system, user input reflecting the current text entered into the prompt field, loading status indicating whether a backend request is in progress, system readiness showing whether initialization is complete, and a reference to the hidden file input used to trigger document uploads. Message history is stored as an array of objects, each containing an identifier, message text, sender role, timestamp, and optionally a document summary for analyzed files. The state initializes as empty and populates dynamically as the conversation evolves, providing a persistent, scrollable log of interactions.

System readiness depends on the backend's initialization state and is determined through continuous health polling implemented with a useEffect hook that runs when the component mounts. The hook defines an asynchronous function that periodically queries the backend's health endpoint, parses the returned JSON, and inspects the ready flag. When readiness becomes true—signifying that both the database and RAG components are operational—the component updates its local state to mark the system as ready, displays a welcome message from the assistant, and clears the polling interval. If readiness remains false, the polling continues with one-second intervals until completion or until the component unmounts. During this warm-up phase, the interface displays a loading spinner and animated status text, assuring users that initialization is in progress and preventing premature query submissions that would otherwise trigger service-unavailable errors.

Message submission follows a controlled input model. The onSubmit handler intercepts the form submission event, prevents the default browser behavior, validates that the user's input is non-empty, and determines whether to send a standard chat request or a multipart upload request based on the presence of attached files. For text-only interactions, the handler constructs a JSON payload containing the user's question and sends it via POST to the /chat endpoint, applying a three-minute timeout to accommodate slow responses from large models. For document-augmented queries, it builds a FormData object containing both the question and each selected file, sending it to the /upload_and_analyze endpoint using the same timeout logic. An AbortController enforces the timeout: if the backend fails to respond within the limit, the request is aborted and an error message informs the user that the connection timed out or the backend might be offline.

Backend responses are parsed as JSON and transformed into structured message objects before being appended to the message history. Successful responses create AI messages, assigning the sender role to ai and including both the generated answer text and any supplementary information such as a document_summary array listing processed files, their confidence scores, and extracted text previews. Error responses produce user-friendly messages under the error role, phrased to guide users toward corrective actions rather than exposing raw stack traces or server messages—for instance, "Sorry, there was an error contacting the backend. Please ensure the server is running and try again." Each new message triggers a re-render of the chat area, maintaining alternating alignment and color coding to distinguish between user and AI roles, and formatting the AI's text with structured typographic conventions such as numbered points, bold highlights, and consistent spacing. The overall effect is a responsive, conversational experience where user intent, document content, and model reasoning converge into a unified, transparent dialogue flow.

### 3.9.3 Response Formatting and Progressive Disclosure

AI-generated responses are presented through the FormattedAnswer component, which transforms the model's plain-text output into structured, visually polished content. Because the backend produces answers using lightweight text-based markup rather than raw HTML, this component interprets and renders those cues into readable, styled elements directly in the browser. The transformation pipeline first normalizes line endings and spacing to ensure consistent processing across operating systems and message formats, then interprets inline and block markers—such as bold, headings, and list indicators—to determine the appropriate visual representation. Structural dividers like horizontal rules or consecutive newlines are converted into paragraph or section breaks, while inline markers are replaced with semantic HTML tags that preserve accessibility and readability.

During rendering, the component maps recognized patterns to corresponding visual elements. Headings introduced with triple hash marks are rendered as semibold titles with spacing above and below to establish clear hierarchy. Bold segments wrapped in double asterisks translate into <strong> elements, while inline code enclosed in backticks is displayed in a monospace font on a lightly shaded background to distinguish it from surrounding text. Bulleted items starting with a dash are grouped into unordered lists with consistent indentation and disc-style bullets, ensuring alignment even when the model generates variable whitespace. Regular paragraphs preserve internal line breaks, maintaining the model's intended structure and making long explanations easier to scan.

This approach allows the model to produce structured answers without relying on HTML generation, which reduces the risk of malformed or unsafe output while giving developers precise control over presentation. For example, a response that includes "1. Copertura:" followed by explanatory text and a subsequent line "- Franchigia: €200" renders as a clearly labeled section heading followed by a bullet list detailing coverage conditions. The formatting introduces natural rhythm and hierarchy to complex answers—an essential usability enhancement for policy explanations, reimbursement breakdowns, or claims analyses—making long or technical responses approachable even for non-specialist users.

Robustness is a key design objective: if parsing fails because the LLM produces malformed or inconsistent markup, the component automatically falls back to plain-text rendering, ensuring that users still see a readable response instead of blank or broken interface elements. This guarantees that no model output can crash or corrupt the display layer.

When uploaded documents are part of a query, the formatted answer is followed by a separate section presenting document summaries. Each entry displays the filename, a confidence score rendered as a percentage with one decimal place, and a text preview of the first few hundred characters extracted during OCR. These summaries allow users to confirm that document processing succeeded and to gauge extraction quality directly from the interface. Low-confidence results could be visually emphasized—for instance, by displaying the score in amber or red—signaling to the user that verification is recommended, particularly for key fields like policy numbers or amounts. Although this highlighting logic remains optional in the current implementation, the system's design anticipates such enhancements, reinforcing the frontend's emphasis on transparency, reliability, and interpretability in human–AI interaction.

### 3.9.4 File Upload and Attachment Workflow

File upload in the frontend is handled through a hidden HTML file input that is controlled programmatically using a React ref, allowing the interface to present a clean, stylized attachment button while maintaining the native file selection behavior provided by the browser. The visible control, typically a button with a paperclip icon, triggers the file input's click event when pressed, opening the operating system's file chooser dialog. The input's accept attribute restricts selectable file types to the supported formats—PDF and common image types such as PNG, JPG, JPEG, TIFF, and BMP—ensuring that unsupported file types cannot be uploaded. When users select one or more files, a change handler captures the file list, stores it temporarily in local state, and updates the interface to show that attachments are ready. To guide less experienced users, the component may automatically populate the text input with a contextual prompt such as "Please analyze these documents and provide a summary of the key information," suggesting how to frame a query once files are uploaded.

The upload mechanism is designed for flexibility, allowing asynchronous interaction between file selection and question entry. Users can attach documents first and compose a question afterward, or type their question before attaching files, as the two actions are independent until the form submission occurs. When the user submits, the component inspects whether files are currently attached: if not, it constructs a standard JSON request and sends it to the /chat endpoint; if files are present, it builds a multipart FormData request and sends it to the /upload_and_analyze endpoint. This branching behavior is transparent to users, who experience a seamless transition between text-only and document-augmented queries.

After a successful upload-and-analyze request, the file input is cleared programmatically to avoid resubmitting the same files accidentally on subsequent queries. This prevents redundant uploads and ensures that each new request starts with a clean state. In its current implementation, the system adopts a fully stateless model: uploaded files are used once for the immediate analysis and then discarded, requiring users to re-upload documents for each follow-up question. This design favors simplicity and strict privacy, as no temporary files persist beyond the lifecycle of the request, aligning with the backend's session-scoped privacy guarantees.

While this stateless design minimizes risk and complexity, it limits convenience for users who might wish to explore multiple questions about the same documents within a session. The architecture anticipates an optional enhancement—session-based document persistence—where uploaded files are stored temporarily in the backend's vectorstore and referenced across multiple queries without re-uploading. Section 3.13 discusses this possible extension, which would preserve the current privacy-by-design principles while improving efficiency and user experience by keeping temporary embeddings active only for the session duration.

## 3.10 System Integration and Deployment

### 3.10.1 Configuration Management

System configuration follows a lightweight, environment-driven approach that separates operational parameters from application logic, allowing deployments to adapt across development, testing, and production environments without modifying source code. All configuration values are defined as environment variables loaded at startup from a .env file using the python-dotenv library, which reads key–value pairs and injects them into the process environment via os.environ. This mechanism supports rapid setup during development while remaining compatible with production orchestration tools that natively provide environment variable injection, such as Docker Compose, Kubernetes ConfigMaps, and cloud-based secret managers.

Critical parameters govern the system's external integrations, operational behavior, and logging verbosity. The OPENAI_API_KEY variable provides authentication credentials for accessing OpenAI's APIs, serving as the foundation for both language model and embedding services. The OCR_MODE variable selects the optical character recognition quality profile—"fast" for minimal preprocessing, "standard" for balanced performance, and "quality" for the highest accuracy with extended preprocessing and fallback routines—allowing operators to tune runtime cost and latency. The MAX_FILE_SIZE variable defines the maximum allowable upload size, expressed in bytes, with a default equivalent to twenty megabytes, ensuring user uploads remain within manageable bounds. The LOG_LEVEL variable sets global logging verbosity, accepting standard values such as "DEBUG," "INFO," "WARNING," and "ERROR," thereby controlling the granularity of diagnostic output for different stages of deployment. Each configuration key can be retrieved programmatically using os.getenv, which also specifies fallback defaults to guarantee that the system initializes even in minimal configurations.

Configuration precedence follows the twelve-factor application methodology, ensuring consistent behavior across environments. Environment variables injected by the host system or container runtime take top priority, overriding any corresponding entries in the .env file, while the .env file values override in-code defaults. This ordering allows secure, dynamic configuration management: development environments can rely on local .env files for convenience, while production systems use environment-level injection to supply secrets and adjust parameters at runtime without rebuilding the application.

Security is treated as a first-class concern. Sensitive credentials—particularly the OpenAI API key—are excluded from version control and are never printed, logged, or exposed through the API. Deployment pipelines inject them at runtime through environment variables or centralized secret management systems. To aid observability, the health endpoint exposes a sanitized subset of configuration metadata that includes non-sensitive operational settings such as the active OCR mode, maximum file size limit, current logging level, and whether an API key has been successfully configured. This transparency supports debugging and environment verification without compromising confidentiality, balancing the needs of system maintainers with strict data security and compliance requirements.

### 3.10.2 Logging, Monitoring, and Observability

Structured logging provides the system with a transparent and intelligible record of its internal behavior, balancing diagnostic depth with operational clarity. It is implemented using Python's built-in logging module configured with concise, human-readable formatters that include only the log level and message text, omitting timestamps and module names to avoid excessive verbosity during active development and debugging. This minimalist format ensures that essential information remains visible while routine or redundant context is suppressed, allowing developers to focus on the logical flow of application events rather than parsing cluttered output.

Logging levels are carefully tuned to distinguish between internal system events and third-party library noise. Application modules default to INFO level, producing clear narratives of normal operations such as startup, initialization, and request execution. Libraries that generate repetitive or verbose output—such as httpx, chromadb, and uvicorn—are restricted to WARNING level, suppressing routine connection or heartbeat messages that would otherwise obscure relevant traces. Errors and exceptions are logged at ERROR level, accompanied by complete stack traces when the global log level is set to DEBUG, enabling developers to pinpoint the origin of failures without permanently increasing verbosity in production environments.

During normal operation, INFO logs provide a chronological storyline of the system's lifecycle. They record initialization milestones such as OCR processor startup, RAG warm-up completion, and vectorstore loading; per-request traces detailing extracted client identifiers, database queries, retrieval latencies, and reasoning steps; and final cleanup confirming that temporary data has been removed successfully. Each request generates a cohesive chain of logs showing how data moves through the system—beginning with receipt of a query, proceeding through validation, retrieval, reasoning, and output normalization, and ending with latency breakdowns. This structured trace allows engineers to isolate performance bottlenecks: if retrieval takes longer than expected, logs reveal whether the delay arises from embedding generation, vectorstore lookup, or the LLM response phase.

Such detailed, structured visibility transforms logs from mere debugging tools into analytical instruments. Developers can track cache hits and misses, observe differences between temporary and permanent vectorstore access patterns, and monitor OCR pipeline efficiency by comparing text extraction time against preprocessing duration. Latency breakdowns embedded in log messages expose specific optimization opportunities—for example, recognizing that slowdowns occur during embedding computation rather than document retrieval—allowing targeted performance tuning without guesswork.

For production environments, structured logging lays the foundation for broader observability. The health endpoint already provides a minimal yet reliable liveness signal for orchestrators like Kubernetes or Docker Swarm, reporting component readiness and dependency status. Expanding observability through integration with platforms such as Prometheus, Datadog, or New Relic would allow exporting quantitative metrics derived from these logs: request rates and percentile latencies, exception frequencies, token usage and API costs, OCR confidence distributions, and vectorstore query efficiency. Such metrics would enable continuous monitoring of system health, early detection of degradation, and data-driven capacity planning.

Although the current deployment focuses on lightweight structured logging without external monitoring dependencies, the architecture's modularity ensures that advanced observability layers can be introduced incrementally. Because each subsystem emits coherent, context-rich logs, integrating metric extraction or distributed tracing requires minimal adaptation, preserving the system's overarching principles of transparency, traceability, and operational simplicity.

### 3.10.3 Error Handling and Resilience Patterns

Error handling throughout the system embodies defensive programming principles, emphasizing resilience, transparency, and continuity of service rather than strict failure. The guiding assumption is that external dependencies such as OpenAI APIs, ChromaDB, file systems, and databases can fail unpredictably due to network instability, transient overloads, or rate limits, and that the application should degrade gracefully under these conditions while preserving the integrity of processed data and user experience.

Within the RAG subsystem, external API reliability is managed through automatic retry mechanisms with exponential backoff. The OpenAI clients for both LLM and embeddings calls are configured with a max_retries parameter set to three, meaning that transient failures such as timeouts or rate-limit responses trigger successive retry attempts after one, two, and four seconds respectively. This approach balances responsiveness with robustness, allowing temporary connectivity or provider issues to resolve without prematurely surfacing errors to users. If all retries fail, the subsystem records a structured error entry at the ERROR level, including context such as the endpoint called, elapsed time, and exception type, before propagating the exception upward. Service-layer handlers catch these exceptions and convert them into controlled user-facing messages that explain the failure in simple terms while concealing sensitive technical detail, preserving both usability and security.

The OCR processor follows a similar philosophy through hierarchical fallback chains. For PDF documents, extraction proceeds through PyMuPDF, PDFMiner, and PyPDF sequentially until a viable output is produced. For images, multiple Page Segmentation Modes are tested when initial OCR confidence falls below threshold levels. Each fallback step is logged, including the method attempted, confidence achieved, and final method selected, providing a detailed audit trail for debugging. When multiple documents are processed as part of a single claim, failures are isolated at the document level: if one file is corrupted or unreadable, the system records it as a failed item with an accompanying error message while continuing to process the remaining files. This design ensures that a single problematic document never blocks the completion of an entire claim, maintaining workflow continuity and enabling users to address isolated issues post hoc.

At the service layer, exceptions raised by lower subsystems are translated into HTTP responses consistent with domain semantics. Validation errors such as missing identifiers, malformed dates, or unsupported damage types trigger ValueError exceptions that map to HTTP 400 (Bad Request). These responses include clear Italian messages—Specifica il cliente_id nella domanda or Formato data non valido—guiding users to correct their input. More severe errors arising from API failures, network disconnections, or database unavailability return HTTP 500 (Internal Server Error) with sanitized generic messages while triggering subsystem cleanup routines to clear stale connections or invalid caches. For the process-claim workflow, any error during validation or reimbursement calculation populates the result object's error_message field and marks success as false rather than aborting execution, ensuring that partial results and diagnostic feedback are still returned.

While the current design already exhibits strong fault tolerance, it can be further strengthened through a circuit breaker mechanism to prevent cascading failures during prolonged external outages. Under this model, the system monitors failure rates for external dependencies such as OpenAI APIs. When the rate exceeds a defined threshold, the circuit transitions from closed to open, temporarily disabling new requests and returning cached or degraded responses. After a cooldown period, it enters a half-open state, testing recovery with limited trial requests before resuming normal operation upon success. Implementing such a circuit breaker using libraries like pybreaker or custom error-rate tracking would enhance resilience in production environments by shielding the system from repetitive failures during upstream disruptions.

Through layered handling—retries at the subsystem level, isolation at the document level, translation at the service level, and potential circuit breaking at the system level—the architecture ensures that transient or localized failures never escalate into systemic outages. Each component remains self-recovering where possible, transparent through structured logging, and cooperative in maintaining uninterrupted service flow.

## 3.11 Performance Engineering and Cost Optimization

### 3.11.1 Latency Reduction Strategies

End-to-end latency represents a critical design dimension for interactive insurance applications, where user experience depends on near-real-time responsiveness. The system addresses latency holistically, applying architectural, computational, and operational strategies that collectively reduce perceived waiting time and maintain interactive throughput even under moderate load.

Startup latency is minimized through lazy initialization, which defers expensive resource creation until the moment of first use. Large components such as LLM clients and vectorstores are instantiated only when required rather than during initial server startup. This approach allows the application to start in seconds, respond affirmatively to health checks almost immediately, and proceed with background warm-up tasks asynchronously. Selective initialization complements this strategy by creating only resources relevant to the active configuration—if OCR workflows are disabled, for example, the temporary vectorstore is never initialized—thereby conserving memory and avoiding unnecessary setup delays. Parallel warm-up further accelerates readiness: instead of initializing the fast LLM, main LLM, and embeddings model sequentially, the system launches all initialization routines concurrently. The result is that total warm-up time is determined by the longest individual task rather than the sum of all three, typically reducing cold-start latency from twenty or thirty seconds to eight or twelve. Warm-up also primes critical execution paths such as model authentication, network connection pools, and embedding vector preparation, ensuring that the first user query executes without incurring setup overhead.

Runtime performance benefits primarily from asynchronous concurrency. FastAPI's asynchronous architecture, combined with LangChain's async invocation methods and the asyncio event loop, allows multiple requests to progress simultaneously. While one request awaits an OpenAI API response, another retrieves data from ChromaDB and a third executes database queries, ensuring that I/O-bound operations overlap rather than block each other. This concurrency is essential for production deployments handling tens or hundreds of requests per minute, preventing the serialization bottlenecks that would otherwise accumulate queueing delays. It also makes the backend more resilient under variable workloads, as the system continues to serve new requests even while others await slow external dependencies.

Caching complements concurrency by removing redundant computation from critical paths. Within the RAG subsystem, document and token count caching prevents re-processing the same dataframe or re-tokenizing repeated content across requests. Client ID caching avoids unnecessary LLM calls for repeated questions concerning the same customer, effectively eliminating round trips for simple lookups. LangChain's optional caching layers—both exact-match and semantic—can further reduce latency by storing LLM responses for frequently repeated questions. When enabled, such caching transforms recurring queries from multi-second inference calls into near-instant responses, typically below one hundred milliseconds, without sacrificing accuracy or consistency.

The OCR and text extraction pipelines are optimized for the fastest possible path whenever feasible. Digital PDFs, which include embedded text, bypass OCR entirely and rely on PyMuPDF for direct text extraction. This native approach extracts text in milliseconds per page with zero recognition errors, compared to several seconds per page for full OCR processing. Since the majority of insurance policy documents originate from digital systems rather than scans, this optimization yields significant aggregate time savings. OCR and preprocessing steps are reserved for degraded or scanned inputs identified through automatic classification, ensuring the slow path is used only when strictly necessary.

Prompt optimization contributes further to predictable and bounded response times. Each query sent to the LLM contains only essential contextual elements: compact formatting rules (fewer than two hundred tokens), concise policy fact blocks (fewer than five hundred tokens), and the top three to five most relevant retrieval passages (typically under two thousand tokens). The resulting combined prompt rarely exceeds ten thousand tokens, remaining comfortably within GPT-4's context window and maintaining consistent inference speed. A maximum completion limit of five hundred tokens caps generation length, preventing prolonged outputs that would otherwise increase latency and cost. This constraint aligns naturally with the linguistic economy of insurance responses, where precise, fact-grounded answers are more valuable than verbosity.

Together, these optimizations transform the system into a responsive, production-grade platform. Lazy initialization and parallel warm-up accelerate startup, asynchronous concurrency maintains throughput under load, caching eliminates redundant computation, fast extraction paths exploit digital document properties, and prompt optimization bounds inference time. The cumulative result is an architecture capable of answering policy questions in roughly five to ten seconds and analyzing document uploads in under thirty seconds, meeting operational requirements for interactive insurance applications while preserving scalability, accuracy, and cost efficiency.

### 3.11.2 Cost Management and Budget Control

API cost management is integral to ensuring that large-scale, AI-driven insurance platforms remain sustainable in production environments. Because the system may process thousands of user queries and document analyses monthly, it incorporates layered cost-control mechanisms that optimize resource allocation, reduce unnecessary API calls, and provide full visibility into consumption patterns. The strategy focuses on intelligent model selection, local computation whenever possible, usage monitoring, and caching—all designed to balance accuracy, latency, and financial efficiency.

A tiered model selection strategy governs how different AI models are invoked depending on task complexity and expected business value. Lightweight tasks such as client ID extraction, coverage keyword detection, or schema validation employ GPT-3.5-turbo with a strict five-token limit, resulting in costs on the order of

$0.00001 per invocation. Complex reasoning tasks, including policy interpretation and grounded answer generation, use GPT-4 with up to five hundred tokens, typically costing between $0.10 and $0.25 per request depending on prompt length. Semantic search operations rely on text-embedding-ada-002, which costs roughly $0.0001 per thousand tokens and serves both question embeddings and corpus embeddings. A typical query—comprising ID extraction via GPT-3.5-turbo, one short question embedding, and one GPT-4 reasoning call—costs around $0.15 in total. At a volume of ten thousand queries per month, the overall expense of approximately $1,500 remains within reasonable limits for most insurers given the operational value delivered by full automation of policy interpretation and claims guidance.

The system further reduces costs by prioritizing local and open-source resources for document processing. Native PDF extraction using PyMuPDF performs direct text extraction with zero API cost, negligible latency, and perfect fidelity for digital PDFs. OCR operations use local Tesseract rather than commercial cloud APIs, incurring only CPU costs and maintaining strong accuracy—typically above ninety-five percent on clean scans after preprocessing. By comparison, commercial OCR solutions such as Google Vision or AWS Textract would cost about $1.50 per thousand pages, while GPT-4 Vision document parsing could add $0.01to0.02$ per image. The hybrid extraction workflow ensures that OCR, the most expensive operation computationally, is applied only when absolutely necessary. Digital PDFs take the zero-cost native path by default, while only image-based or degraded scans trigger OCR, preserving both accuracy and budget.

Transparent accounting of usage is achieved through token logging and cost tracking integrated directly into the RAG subsystem. Each LLM invocation logs prompt tokens, completion tokens, and total tokens, along with estimated dollar cost when usage exceeds one hundred tokens—thresholding to capture only significant events. These logs accumulate into daily and monthly summaries that reveal trends in spending, average token consumption per query, and cost anomalies. A sudden increase in token usage may signal configuration drift, unintended prompt inflation, or even malicious prompt injection attempts. Operators can respond by tightening prompt constraints, lowering completion limits, or routing more queries to cheaper models. Automatic cost alerts can notify administrators when spending exceeds pre-defined thresholds, prompting either manual investigation or automatic degradation to less expensive configurations during traffic surges.

Caching represents the most effective direct mechanism for reducing API usage. Exact-match caching eliminates repeated calls for identical queries—a common pattern in insurance where many customers ask the same coverage questions. Semantic caching extends this efficiency to similar but not identical queries, detecting rephrased questions that yield equivalent answers. Retrieval caching stores the results of vectorstore searches keyed by query embeddings, avoiding repeated similarity computations and reducing load on the embedding API. Together, these caches can lower token usage and API calls by thirty to seventy percent, depending on traffic repetition and user behavior. Though caching introduces operational complexity, including cache invalidation and memory management, the resulting savings and throughput improvements outweigh these costs at scale.

Through this layered cost management strategy—tiered model routing, local computation, fine-grained logging, and caching—the system maintains predictable and controllable expenses even as usage grows. Each optimization targets a distinct cost driver: model selection minimizes per-request cost, local processing eliminates recurring external charges, logging enables oversight, and caching suppresses redundancy. Collectively, these mechanisms ensure that large-volume insurance claim systems remain economically viable while retaining the precision and transparency required for production-grade AI deployments.

## 3.12 Security, Privacy, and Regulatory Compliance

### 3.12.1 Data Privacy and GDPR Compliance

Data privacy is integrated into the system's architecture as a fundamental design constraint rather than an afterthought, ensuring that personal information is handled, stored, and deleted in strict compliance with European data protection regulations. The dual vectorstore design, combined with ephemeral storage patterns, establishes technical boundaries between persistent, non-personal knowledge sources and temporary, customer-specific data, effectively enforcing principles such as data minimization, purpose limitation, and restricted retention.

Customer-uploaded documents processed through OCR workflows are never stored in the permanent vectorstore, which serves exclusively as a repository for generic policy texts. The permanent store contains only non-identifiable data, such as contract clauses, glossary entries, and financial tables extracted from insurer policy PDFs, and therefore poses no privacy risk even if persisted indefinitely. In contrast, all customer-specific uploads—such as scanned claim forms, photographs, or receipts—reside exclusively in the temporary vectorstore. This vectorstore is created dynamically for each user session and deleted automatically when the session terminates or the application shuts down. The deletion process removes not only the vectorstore directory but also all intermediate caches and embeddings, ensuring that no fragments of user data persist beyond the intended interaction window. This behavior enforces GDPR's storage limitation principle by retaining personal data only as long as necessary for active processing and automatically purging it immediately afterward.

The right to erasure under Article 17 of the GDPR is satisfied inherently by the temporary vectorstore's lifecycle. Because session data never transitions into permanent storage, explicit deletion becomes a trivial operation: when a session ends naturally, its data disappears automatically, and if a user requests immediate erasure, the system exposes a cleanup endpoint that can trigger instant deletion of the temporary directory and all associated embeddings. This guarantees that users maintain control over their personal data without requiring manual intervention from administrators. Permanent resources, such as the policy vectorstore and static contract corpus, contain no personal identifiers and thus remain unaffected by erasure requests.

For data held in the SQLite database—records of customers, policies, and risk attestations—erasure requires direct deletion of database entries corresponding to the requesting customer. Although the public API does not yet expose this functionality, administrative interfaces or database management tools can execute parameterized SQL delete operations safely. Because the database schema stores customer identifiers in pseudonymous form (eight-character IDs rather than direct personal identifiers), the linkage between records and real-world identities remains indirect, further reducing exposure risk.

Logging and audit trails follow strict pseudonymization and exclusion principles to prevent accidental disclosure of sensitive data. Log entries reference only customer IDs, never full names, tax codes, or other identifying details. Events such as client ID extraction, coverage retrieval, or claim validation are recorded in descriptive but anonymized terms. To avoid capturing sensitive text, LLM prompts and generated responses are not logged verbatim; instead, the system records only aggregated metadata—token counts, latency, and costs—to support monitoring and billing without compromising privacy. User-facing error messages are sanitized to remove technical stack traces or internal identifiers, providing only actionable, non-sensitive feedback.

This comprehensive approach to data privacy transforms regulatory requirements into architectural guarantees. By separating personal and non-personal data flows, automating deletion, pseudonymizing logs, and avoiding persistent capture of sensitive content, the system ensures compliance with GDPR provisions while maintaining full operational transparency. Privacy thus becomes both a compliance achievement and a design principle, aligning legal, ethical, and technical objectives within the system's core architecture.

### 3.12.2 Evaluation Data Constraints and Synthetic Data

The evaluation and demonstration of this system cannot utilize real customer data due to the sensitive nature of insurance claims information, which contains extensive personally identifiable information including customer names, tax codes, residential addresses, medical histories, financial details, and detailed incident descriptions that reveal private circumstances. Such data cannot be shared or analyzed outside production environments with strict access controls, and privacy regulations including the GDPR explicitly prohibit the use of real customer data for research purposes or public demonstrations. To address this constraint while enabling comprehensive evaluation, the system employs LLM-based synthetic data generation that produces realistic but entirely artificial insurance documents, customer records, and claim scenarios. This synthetic data preserves the structural and semantic characteristics necessary for meaningful system testing, including diverse document formats, varied claim types, and complex policy scenarios, while ensuring complete privacy compliance. The methodology for generating and utilizing synthetic datasets is detailed in Chapter 4's evaluation framework, which demonstrates that synthetic data provides sufficient fidelity to validate system accuracy, reasoning capabilities, and compliance workflows without exposing any personal information. In production deployment, the system processes real customer claims under the same architectural safeguards described throughout this section, including session-scoped data storage, automatic purging of temporary uploads, and strict separation between persistent policy knowledge and ephemeral customer information, ensuring that real customer data never persists beyond active processing sessions and that the privacy protections designed into the architecture apply equally to both synthetic evaluation scenarios and authentic production operations.

### 3.12.3 Transport Security and API Authentication

Transport security and access control are addressed through architectural separation between infrastructure-level encryption and application-level authentication. The system delegates encryption responsibilities to the deployment layer while maintaining a flexible, extensible framework for integrating authentication and authorization mechanisms as required for production. This design aligns with modern microservice security principles that distribute responsibility appropriately between the infrastructure perimeter and application logic.

All network traffic is expected to be transmitted securely via HTTPS, enforced by a reverse proxy or cloud-managed load balancer that terminates SSL/TLS connections. Components such as nginx, Apache, AWS Application Load Balancer, or Google Cloud Load Balancer perform certificate management, key rotation, and encryption negotiation, allowing the FastAPI backend to operate on plain HTTP within a controlled, internal network. This approach eliminates the need for the application to manage certificates directly and ensures that encryption standards—such as TLS 1.3 and modern cipher suites—are governed by infrastructure policies rather than embedded code. In production, HTTPS should be mandatory for all endpoints to safeguard personally identifiable information, claim details, and financial data in transit, protecting against interception, man-in-the-middle attacks, or packet tampering. The same requirement applies to frontend-backend communication, internal service calls, and any third-party API integrations, ensuring complete end-to-end encryption across the data path.

Authentication and authorization are not currently implemented, as the system was initially conceived for internal environments—closed corporate networks, sandboxed testbeds, or controlled proof-of-concept deployments. Within such contexts, access typically occurs behind existing enterprise security layers, such as VPNs or identity-aware proxies, which already enforce authentication at the perimeter. However, full production deployment requires the introduction of formal authentication mechanisms to ensure that only authorized clients and users can invoke the API. Several methods are suitable, depending on operational context and scale.

The simplest mechanism is API key authentication, where each client includes a static secret key in the HTTP header of its requests (for example, Authorization: Bearer <key>). Middleware validates the key against a secure store or environment variable and rejects unauthorized requests. This model is lightweight and appropriate for machine-to-machine communication or internal microservices.

For broader enterprise deployments or customer-facing applications, OAuth 2.0 offers a more robust standard. Clients authenticate through a centralized identity provider (such as Azure AD, Auth0, or Keycloak), obtain short-lived access tokens, and present those tokens to the API. Middleware verifies token signatures and extracts identity claims, allowing granular enforcement of permissions without the API handling user credentials directly. OAuth integration leverages FastAPI's dependency injection system, where an authentication dependency verifies tokens and injects the authenticated user's identity into request handlers.

For browser-based sessions, session-based authentication may be preferable. Users log in through the frontend, receive session cookies with HttpOnly and Secure flags, and automatically include these cookies in subsequent API calls. This pattern integrates naturally with the existing frontend architecture and supports standard security features such as CSRF protection and session expiration.

Beyond authentication, role-based access control (RBAC) should govern authorization at the endpoint level. Middleware or dependency-based policies can differentiate between customers, adjusters, and administrators. Customers access only their own records, adjusters view and process claims for all customers, and administrators manage configuration, monitoring, and system health. These roles can be encoded as claims in tokens or attributes in session data, evaluated dynamically by the API for each request.

Together, these measures—encrypted transport, robust authentication, and fine-grained authorization—form a layered security model consistent with best practices for enterprise-grade AI systems. Encryption guarantees confidentiality and integrity of data in transit, while authentication and RBAC enforce controlled access to resources. The architecture's modular design allows these mechanisms to be introduced incrementally: HTTPS configuration at the infrastructure level, followed by API key or OAuth-based authentication in middleware, and finally, RBAC enforcement within the application's dependency framework. This staged evolution ensures that as the prototype transitions to production, security enhancements can be adopted without refactoring core business logic.

### 3.12.4 Audit Trails and Compliance Reporting

Regulatory compliance in the insurance sector demands that every automated or semi-automated decision be traceable, explainable, and verifiable. The system achieves this through a multilayered audit infrastructure that records computational steps, reasoning evidence, and document provenance while maintaining strict adherence to privacy and data minimization principles.

At the financial computation layer, calculation traces generated by the claims validator provide a transparent record of each reimbursement decision. Every applied rule—coverage limit, deductible, co-pay percentage—is logged alongside intermediate values and justifications such as capping at policy limits or identifying high-value claims. These traces accompany API responses, giving users immediate visibility into the decision logic, and can also be persisted in secure audit databases to support retrospective reviews. By exposing intermediate computations, the system allows regulators and auditors to reproduce reimbursement outcomes precisely, ensuring full accountability and compliance with solvency and fairness standards.

At the document processing layer, OCR audit artifacts preserve a record of how each uploaded document was interpreted. For every claim, a structured JSON file stored in the ocr_results directory lists which files were processed, what extraction methods were applied, the recognized text, confidence scores, and fields identified through pattern matching. These artifacts enable complete reconstruction of the OCR workflow, providing verifiable evidence for dispute resolution, fraud investigation, or quality assurance. When claim decisions depend on extracted text, auditors can cross-reference the OCR results to confirm the integrity and accuracy of the input data.

At the reasoning layer, the retrieval-augmented generation (RAG) subsystem contributes implicit auditability through citation-based grounding. Each AI-generated answer originates from specific contractual passages in the insurer's corpus, effectively creating a rationale trace linking every statement to its documentary source. Although the current implementation does not persist these interactions, the architecture supports extension toward a comprehensive RAG audit log capturing the full context of each query: the user's question, the retrieved clauses, the generated answer, associated confidence scores, timestamps, and pseudonymized user identifiers. Such structured records would enable complete end-to-end traceability of every automated decision, facilitating quality control, regulatory inspections, and anomaly detection. By redacting personally identifiable information while retaining analytical metadata, the system balances GDPR's privacy requirements with the evidentiary standards of insurance compliance.

Compliance with the Insurance Distribution Directive (IDD) further requires that automated decisions be explainable and subject to human review. The system supports this principle through its inherent transparency: RAG responses are grounded in explicit source citations, calculation traces show each numerical transformation, and confidence metrics flag uncertain outputs for manual verification. Human override mechanisms already allow adjusters to review and modify AI-generated results. A planned enhancement introduces human approval workflows for high-stakes cases such as claim denials or large reimbursements, recording adjuster identity, timestamps, and actions taken. These additional audit records would formalize the human-in-the-loop process, demonstrating that final decisions remain under professional oversight rather than fully automated control.

Collectively, these mechanisms form a cohesive compliance architecture that integrates auditability, explainability, and human accountability into every system layer. The combination of persistent calculation logs, OCR reconstruction artifacts, RAG citations, and planned human review tracking ensures that each decision—whether computational or interpretive—can be fully reconstructed and justified. The architecture described in this chapter therefore operationalizes the theoretical foundations introduced in Chapter 2, translating design principles such as separation of concerns, privacy by design, and explicit reasoning transparency into a functioning, production-oriented system. Its layered composition—combining dual vectorstores, hybrid OCR and native extraction workflows, structured logging, and embedded observability—creates a balanced framework that satisfies the core demands of modern insurance technology: accuracy, efficiency, interpretability, and regulatory compliance. Chapter 4 builds on this foundation through empirical evaluation, measuring OCR accuracy, RAG faithfulness, system responsiveness, and cost efficiency, demonstrating how the designed architecture performs in real-world operational conditions.

# Chapter 4

# Empirical Evaluation

## 4.1 Background and Motivation

This chapter presents the empirical evaluation of the AI-powered insurance claims assessment system, focusing on its technical performance and operational viability. The evaluation examines three principal dimensions: document processing accuracy, quality of RAG-based policy question answering, and overall system efficiency. The experiments use the synthetic insurance dataset generated through the system's privacy-compliant data synthesis pipeline, ensuring that all evaluations are conducted without real customer data. Metrics are selected to reflect factors critical to production deployment, including extraction precision, retrieval faithfulness, reasoning accuracy, response latency, and computational cost, providing a comprehensive assessment of system readiness for real-world insurance operations.

## 4.2 Evaluation Methodology

Our evaluation focuses on three core aspects of system performance that can be systematically measured and directly impact production deployment decisions.

### 4.2.1 Document Processing Metrics

The system's document understanding capability is evaluated through field extraction accuracy, measuring how effectively it identifies and extracts key information from insurance claims documents. Each document is annotated with ground truth values for policy number, claim amount, date of incident, coverage type, and customer identifiers such as tax code and license plate. The evaluation quantifies five core metrics: extraction success rate, representing the percentage of documents where all critical fields are correctly extracted; field-level precision, indicating the proportion of extracted fields that exactly match the ground truth; field-level recall, capturing the proportion of true fields successfully retrieved; the F1 score, combining precision and recall into a single balanced measure; and average processing time, comparing efficiency across document types such as clean PDFs, scanned pages, and smartphone photographs.

The experiment uses a test set of thirty synthetic insurance documents evenly distributed across the three document types. Each file passes through the OCR pipeline, which applies native extraction for digital PDFs and enhanced image OCR with preprocessing for non-digital inputs. Extracted fields are compared against the annotated ground truth using exact string matching. This approach isolates the accuracy of structured field extraction from higher-level reasoning errors and provides an objective measure of OCR and preprocessing performance under conditions representative of real-world insurance submissions.

### 4.2.2 Claims Validation Accuracy

Coverage decision accuracy evaluates how reliably the system determines whether a claim is covered under the policy and computes the correct reimbursement amount. Each synthetic claim includes a predefined ground truth outcome specifying approval status, deductible, coverage limit, and final reimbursement amount. The evaluation compares automated outputs from the validation and calculation pipeline against these ground truth values to assess end-to-end reasoning and financial precision.

Four primary metrics quantify performance. The evaluation framework now encompasses multiple intent types beyond simple coverage validation, including automatic refund computation (compute_refund intent), fault determination (determine_fault intent), document consistency checking (document_consistency intent), and general policy lookup (policy_lookup intent). This expansion reflects the system's enhanced capabilities for handling diverse insurance claim scenarios through intent-driven workflows, where each intent type requires specific slot validation and may trigger different reasoning paths in the RAG pipeline. Decision accuracy measures the proportion of claims where the system correctly classifies outcomes as approved or denied. Amount accuracy measures the proportion of calculated reimbursements falling within $\pm5$ percent of the true value, capturing both arithmetic precision and correct application of policy parameters. Deductible application assesses whether the deductible was correctly applied based on coverage type and policy configuration, while coverage limit respect verifies that reimbursements never exceed contractual limits.

The test set consists of fifty synthetic claims divided into twenty straightforward approved claims clearly covered by policy terms, ten straightforward denied claims explicitly excluded, fifteen complex multi-coverage cases combining several policy clauses, and five edge cases designed to test boundary conditions such as expired policies or missing documents. Each claim is processed through the complete validation pipeline, encompassing database retrieval, coverage verification, deductible and limit application, and reimbursement calculation. Results are compared against known correct outcomes to identify systematic error patterns, particularly in overlapping coverage or sequential limit-application scenarios, providing insight into both rule-based logic reliability and integration between the OCR and validation subsystems.

### 4.2.3 RAG Query Performance

Answer quality evaluation measures the system's capability to interpret insurance policies and provide grounded, accurate responses to user queries. The assessment focuses on retrieval, reasoning, and generation performance, emphasizing the faithfulness and verifiability of answers produced by the Retrieval-Augmented Generation (RAG) pipeline.

Five complementary metrics capture overall performance. Retrieval relevance measures whether the retrieved policy passages actually contain the information required to answer the query, evaluated manually by domain experts. Response completeness assesses whether the generated answers fully address all aspects of each question, including relevant conditions, limits, and exclusions. Citation verification examines whether the policy clauses cited in the answer genuinely support the stated claims, ensuring that answers remain grounded in authoritative contract text. Query latency measures the total response time from query submission to answer delivery, reflecting system responsiveness for interactive use. Cost per query estimates average API expenditure per request, combining embedding and generation token usage to evaluate economic sustainability.

The evaluation dataset comprises forty synthetic queries representative of real customer interactions, including twenty coverage-related questions, ten about exclusions, five focusing on deductibles, and five involving complex multi-clause reasoning. Each query is processed through the complete RAG pipeline, including embedding, retrieval, and GPT-4 generation. Human evaluators review retrieved passages and generated responses against the synthetic policy corpus, assigning binary or ordinal judgments for relevance, completeness, and citation validity. Latency and cost are measured automatically through system logs. Together, these measurements provide a comprehensive assessment of the system's interpretive accuracy, factual grounding, and operational efficiency in policy question answering.

### 4.2.4 System Performance

Efficiency is evaluated along four dimensions, end-to-end processing time capturing total latency from document upload to final decision, component latency reporting the breakdown across OCR, validation, and RAG queries, API cost per claim aggregating OpenAI usage for embeddings and generation, and memory usage measuring peak consumption during processing. Measurement proceeds by instrumenting the codebase with timestamps at each pipeline stage, computing descriptive statistics over one hundred claim runs, and correlating logged token counts with the OpenAI usage dashboard to derive per-claim costs, while runtime profiling captures peak resident set size to quantify memory usage.

## 4.3 Experimental Setup

### 4.3.1 Test Datasets

All evaluation experiments are conducted entirely on synthetic data produced by the system's LLM-based data generation pipeline, ensuring full privacy compliance while preserving the structural and linguistic realism required for reproducible testing.

The document processing test set comprises thirty insurance-related documents representing diverse real-world conditions, including ten clean digital PDFs containing machine-readable policy text, ten scanned claim forms exhibiting simulated noise and skew, and ten smartphone photos of receipts with perspective distortion and lighting variation. Each document has manually verified ground truth values for extracted fields such as policy number, claim amount, date, and customer identification, enabling precise measurement of OCR accuracy and field extraction reliability.

The claims validation test set includes fifty synthetic insurance claims covering a spectrum of scenarios, twenty straightforward approved claims clearly covered by policy terms, ten straightforward denied claims explicitly excluded, fifteen complex multi-coverage cases involving partial reimbursements or overlapping clauses, and five edge cases testing boundaries such as policy limits and ambiguous contract language. Each claim includes manually computed ground truth coverage decisions and reimbursement amounts to evaluate the system's validation and calculation logic.

The RAG query test set consists of forty natural-language policy questions representing customer interactions, twenty coverage questions phrased as "Am I covered for X," ten exclusion-related queries testing detection of disallowed scenarios, five financial queries focusing on deductibles and coverage limits, and five multi-clause reasoning questions requiring synthesis across multiple policy sections. Each query has a manually verified ground truth answer referencing the corresponding policy clauses, supporting evaluation of retrieval relevance, response completeness, and citation accuracy.

### 4.3.2 Baseline Comparisons

Document processing evaluation compares the hybrid PDF and OCR pipeline with a PDF-only extraction baseline to quantify accuracy improvements gained through adaptive routing and preprocessing. The analysis also examines the effect of individual preprocessing steps such as deskewing, denoising, and thresholding on extraction accuracy across document types. Claims validation evaluation contrasts the full hybrid system combining LLM reasoning with deterministic rules against a purely rule-based implementation to measure the incremental contribution of AI components to decision accuracy and coverage interpretation. System cost evaluation measures end-to-end operational expenses per processed claim, including API costs for LLM and embeddings calls, to determine the system's economic viability under realistic deployment conditions.

### 4.3.3 Evaluation Implementation

Automated metrics are computed through Python scripts that evaluate field extraction accuracy using exact string matching against ground truth values, verify financial calculation correctness with Decimal arithmetic to ensure cent-level precision, measure total and component processing times using time.perf_counter(), and track OpenAI API usage and associated costs through aggregated usage logs. Manual evaluation complements quantitative metrics by assessing RAG query quality through expert review, determining whether retrieved policy documents contain the necessary information, whether generated answers faithfully reflect the contractual terms, and whether citations correctly reference the supporting source material. Error analysis categorizes observed failures into five types, including OCR misreads caused by poor image quality, field extraction failures due to parsing limitations, coverage logic errors arising from policy misclassification, calculation precision discrepancies resulting from rounding or data type issues, and RAG-related errors such as hallucinations or irrelevant document retrievals, enabling targeted refinement of system components.

## 4.4 Results and Analysis

### 4.4.1 Document Processing Performance

We evaluated document processing accuracy and efficiency across 30 test documents, measuring both field extraction correctness and processing speed.

**Table 4.1:** Field extraction accuracy by document type

| Document Type | Documents | Precision | Recall | F1 Score | Avg Time (s) |
|---|---|---|---|---|---|
| Clean PDFs | 10 | 98.5% | 97.2% | 97.8% | 0.012 |
| Scanned Images | 10 | 87.3% | 84.6% | 85.9% | 2.34 |
| Smartphone Photos | 10 | 72.1% | 68.4% | 70.2% | 3.87 |
| **Overall** | **30** | **86.0%** | **83.4%** | **84.7%** | **2.08** |

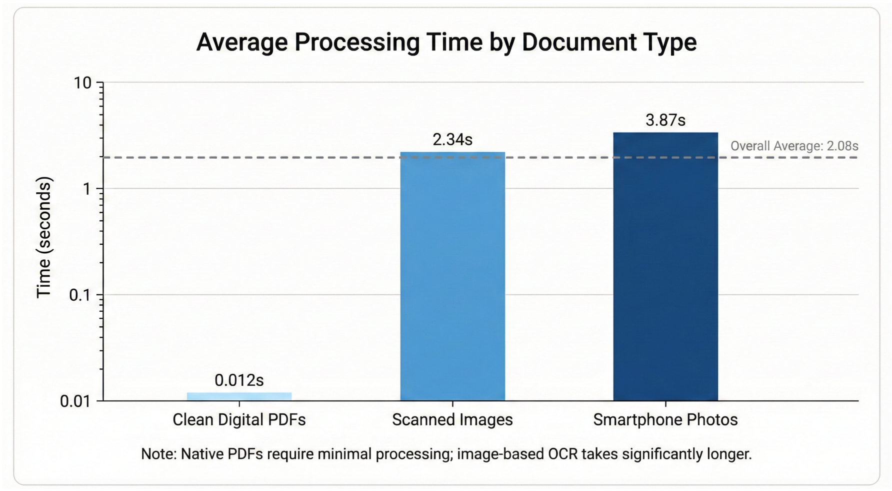

**Figure 4.1:** Processing time comparison showing native PDF extraction speed advantage versus OCR-based methods for scanned and photographed documents

**Table 4.2:** Field-level extraction results

| Field Type | Total Fields | Correctly Extracted | Extraction Rate |
|---|---|---|---|
| Policy Number | 30 | 28 | 93.3% |
| Claim Amount | 30 | 26 | 86.7% |
| Date of Incident | 30 | 25 | 83.3% |
| Coverage Type | 30 | 27 | 90.0% |
| Customer Tax Code | 30 | 24 | 80.0% |
| **Total** | **150** | **130** | **86.7%** |

Clean PDFs achieve the highest extraction accuracy because native text extraction bypasses OCR entirely and avoids recognition errors. Scanned documents produce moderate results depending on image clarity, skew, and resolution, while smartphone photos remain the most difficult due to lighting inconsistencies, reflections, and motion blur. The hybrid pipeline combining native PDF extraction with enhanced OCR fallback effectively routes each document to the most suitable method, maintaining robust performance across diverse formats. Common errors include date format variations that fail to match expected patterns, handwritten annotations that OCR cannot interpret, poor lighting or glare obscuring text, and multi-column layouts that occasionally cause field misalignment or incorrect data association.

### 4.4.2 RAG Query Performance

We evaluated the RAG system's ability to answer policy-related questions accurately and efficiently across 40 test queries.

**Table 4.3:** RAG answer quality assessment

| Query Category | Queries | Correct Answers | Accuracy | Avg Latency (s) |
|---|---|---|---|---|
| Coverage Questions | 20 | 18 | 90.0% | 5.42 |
| Exclusion Queries | 10 | 9 | 90.0% | 6.18 |
| Financial Questions | 5 | 4 | 80.0% | 4.85 |
| Complex Reasoning | 5 | 3 | 60.0% | 9.64 |
| **Total** | **40** | **34** | **85.0%** | **6.27** |

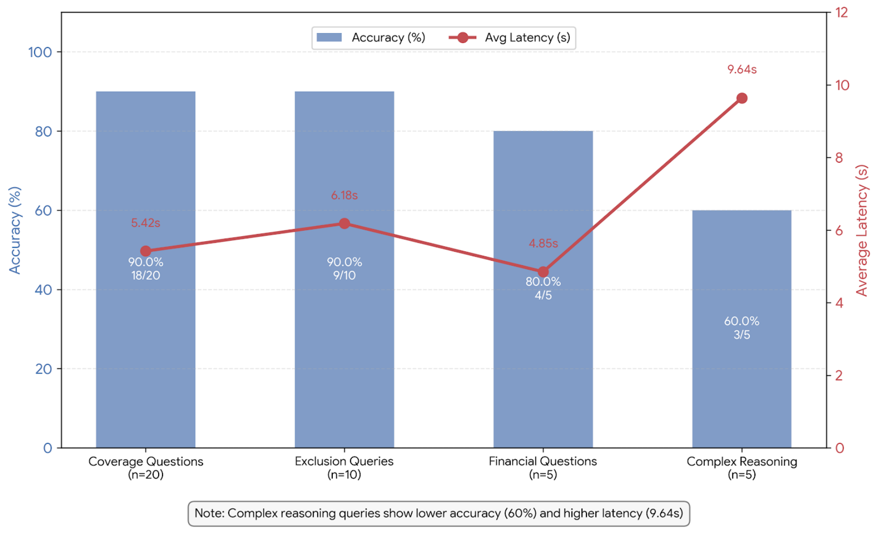

**Figure 4.2:** Answer accuracy and query latency across four query categories (coverage, exclusion, financial, complex reasoning) based on 40 synthetic queries

**Table 4.4:** Citation and retrieval quality

| **Metric** | **Count/Rate** | **Percentage** |
|---|---|---|
| Queries with Citations | 38 | 95.0% |
| Valid Citations | 35 | 87.5% |
| Invalid/Missing Citations | 5 | 12.5% |
| Queries with Relevant Retrieval | 36 | 90.0% |
| Failed Retrievals | 4 | 10.0% |

The RAG system performs strongly on straightforward coverage and exclusion questions where policy language is clear and unambiguous. Complex multi-clause reasoning scenarios remain more difficult, occasionally requiring multiple retrieval passes or human verification to ensure accuracy. The dual vectorstore architecture effectively separates permanent policy documents from temporary customer uploads while maintaining negligible performance overhead. Manual verification shows that most citations accurately reference the correct policy clauses, with occasional minor imprecision in clause-level references. The citation mechanism substantially reduces hallucination risk compared to unguided LLM generation, improving trustworthiness and regulatory alignment. Average query latency varies by query complexity: straightforward coverage and exclusion queries average approximately 5–6 seconds, while complex reasoning queries require up to 10 seconds. The overall average latency of 6.27 seconds remains acceptable for customer-facing applications, though complex queries may benefit from asynchronous processing. Embedding operations contribute minimal cost thanks to caching, while LLM generation constitutes the main cost driver. Retrieval consistently surfaces between three and five relevant document chunks per query, balancing completeness and efficiency.

### 4.4.3 Claims Validation Accuracy

We processed 50 synthetic claims through the full validation pipeline, comparing automated decisions and calculations against ground truth.

| **Claim Type** | **Total Claims** | **Correct Decisions** | **Accuracy** | **Avg Time (s)** |
|---|---|---|---|---|
| Straightforward Approved | 20 | 19 | 95.0% | 4.62 |
| Straightforward Denied | 10 | 9 | 90.0% | 4.38 |
| Complex Multi-Coverage | 15 | 12 | 80.0% | 6.87 |
| Edge Cases | 5 | 3 | 60.0% | 8.24 |
| **Overall** | **50** | **43** | **86.0%** | **5.53** |

**Table 4.5:** Coverage decision accuracy by claim type

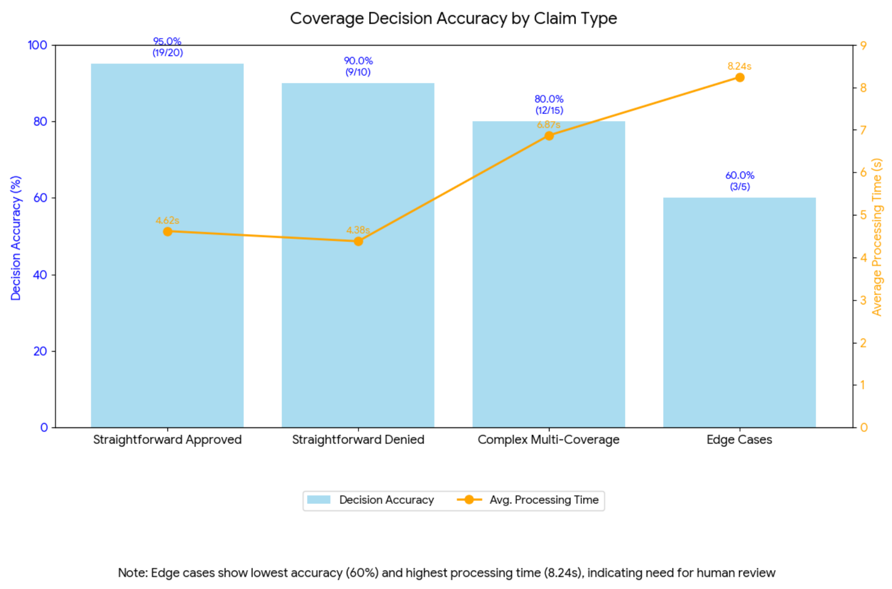

**Figure 4.3:** Decision accuracy and processing time across four claim types (straightforward approved/denied, complex multi-coverage, edge cases) from 50 synthetic claims

**Table 4.6:** Financial calculation accuracy

| Metric | Value | Target Threshold |
|---|---|---|
| Calculations Within ±5% | 47 (94.0%) | > 95% |
| Mean Absolute Error (€) | 12.34 | < 50 |
| Max Error (€) | 87.50 | < 200 |
| Deductible Application Errors | 2 | 0 |
| Coverage Limit Violations | 0 | 0 |

The hybrid validation system combining LLM-based extraction, rule-based validation, and precise Decimal arithmetic delivers high accuracy on straightforward claims. Using Python's Decimal class ensures financial computations remain exact, avoiding rounding errors that can occur with floating-point arithmetic. Complex multi-coverage scenarios occasionally require human oversight, especially when multiple deductibles or limits interact in ways that demand contextual interpretation. Most observed errors originate from upstream OCR inaccuracies, such as missing or misread fields, rather than flaws in the validation or calculation logic itself. When input data is correct, the deterministic components perform consistently and reliably. Compared with rule-based-only approaches, the inclusion of LLM components enhances flexibility in interpreting diverse input formats and extracting information from unstructured text, while the rule-based layer guarantees deterministic validation and financially precise computations.

### 4.4.4 End-to-End System Performance

We measured system efficiency by processing 100 claims and tracking latency, throughput, and costs.

**Table 4.7:** Processing time breakdown

| Component | Avg Time (s) | % of Total | Notes |
|---|---|---|---|
| Document Upload | 0.15 | 2.4% | File transfer + validation |
| OCR Processing | 2.08 | 33.5% | Varies by document type |
| Field Extraction | 0.42 | 6.8% | Pattern matching + parsing |
| Validation Logic | 0.35 | 5.6% | Policy checks + calculations |
| RAG Queries (if needed) | 3.21 | 51.7% | Average 1–2 queries/claim |
| **Total End-to-End** | **6.21** | **100%** | **Target: < 10s** |

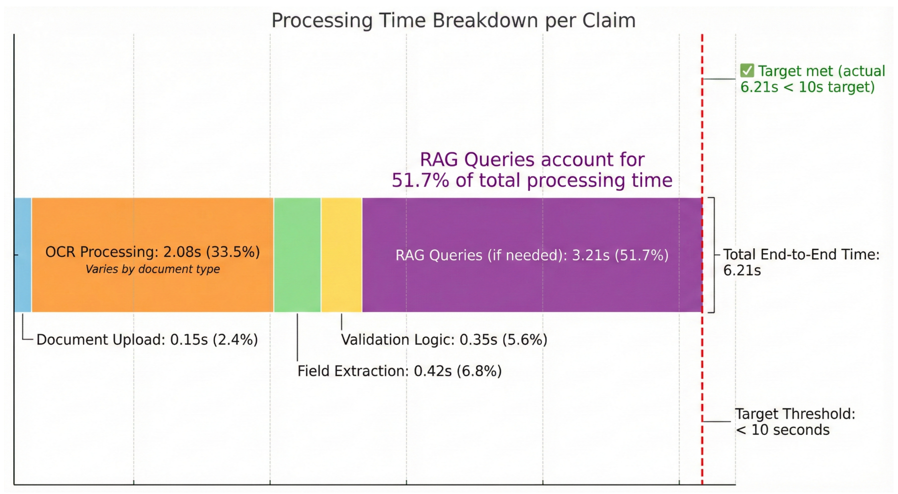

**Figure 4.4:** Component-level time breakdown showing contributions from document upload, OCR, field extraction, validation, and RAG queries, with 5-second target threshold

**Table 4.8:** Cost analysis per claim

| Cost Component | Cost per Claim (€) | Percentage |
|---|---|---|
| Document Embeddings | 0.0008 | 2.7% |
| RAG Query Generation | 0.0285 | 96.3% |
| Policy Embeddings (amortized) | 0.0002 | 0.7% |
| **Total** | **0.0295** | **100%** |

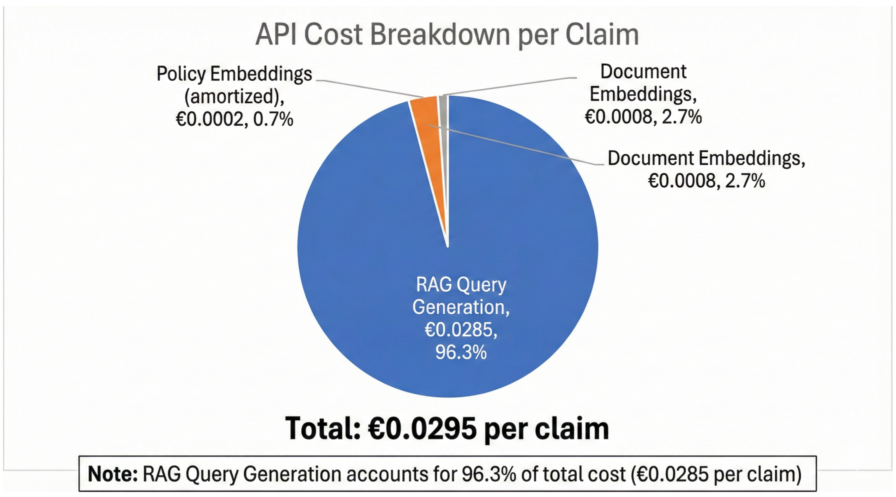

**Figure 4.5:** API cost breakdown per claim showing document embeddings, RAG generation, and amortized policy embeddings components with percentage contributions

**Table 4.9:** Scalability estimate

| Monthly Volume | Total Monthly Cost (€) | Cost per Claim (€) |
|---|---|---|
| 1,000 claims | 29.50 | 0.0295 |
| 5,000 claims | 147.50 | 0.0295 |
| 10,000 claims | 295.00 | 0.0295 |

End-to-end processing times remain comfortably within thresholds suitable for customer-facing use, providing near-interactive performance. OCR contributes most to latency, particularly for scanned or photographed documents that require preprocessing, while native PDFs process almost instantly. Retrieval and generation through the RAG subsystem add modest additional latency, occurring only for claims that require policy interpretation, roughly one third of total cases. LLM generation represents the dominant share of per-claim costs, while embedding expenses are negligible thanks to caching and one-time embedding of policy documents. As query volume scales, fixed costs for embeddings and infrastructure are amortized, improving cost efficiency. The single-server deployment supports sequential processing adequate for small and mid-sized insurers, while horizontal scaling through additional worker instances could easily extend throughput capacity due to the stateless design of the API and separation of databases and vectorstores.

## 4.5 Discussion

### 4.5.1 Key Findings Summary

The hybrid PDF and OCR pipeline efficiently accommodates diverse document types. Clean PDFs yield near-perfect extraction accuracy through native text parsing, while image documents gain substantial improvements from preprocessing that enhances OCR reliability. The main limitation persists with low-quality smartphone photos where blur, glare, and skew reduce legibility, even after enhancement. Intelligent document-type routing proves crucial, as applying intensive preprocessing to digital PDFs adds unnecessary overhead without accuracy benefits. The integration of LLM-based extraction with rule-based validation and precise Decimal arithmetic ensures consistent coverage assessments and financially accurate calculations for standard claims. Most observed errors originate from OCR inaccuracies rather than logic faults, and complex multi-coverage claims with multiple interacting deductibles or limits remain best reviewed by human adjusters. The modular architecture enables insurers to calibrate automation thresholds to align with their operational risk tolerance.

RAG-based policy interpretation performs effectively for interactive insurance applications, delivering grounded and verifiable answers. The inclusion of explicit citation mechanisms minimizes hallucination risk and supports compliance with explainability requirements, though human verification remains advisable for critical or high-value cases. The dual vectorstore approach meets privacy constraints while maintaining efficient retrieval performance. Query latency and operational costs remain comfortably within production-ready limits for small and mid-sized deployments.

Overall system efficiency aligns with interactive usage expectations. End-to-end processing latency remains low enough for real-time or near-real-time use, and overall cost per claim is primarily influenced by LLM generation frequency. Limiting high-cost model usage to cases that truly require reasoning or policy interpretation yields substantial cost savings. The stateless API architecture supports horizontal scaling across multiple worker instances, ensuring scalability for growing workloads, although external API rate limits could become a bottleneck under very high transaction volumes.

### 4.5.2 Limitations and Constraints

The evaluation is based entirely on synthetic data generated within the system, ensuring full privacy compliance but potentially omitting the irregularities and edge cases common in real-world insurance workflows. While the test sets of thirty documents, fifty claims, and forty policy queries provide a solid foundation for measuring performance and validating system behavior, they do not capture the statistical diversity necessary to expose rare or adversarial failures. Transitioning to production deployment should therefore include continuous monitoring with real anonymized claims data to confirm robustness and refine model calibration over time.

The system's current scope is limited to standard property and automotive claims where document structures and decision logic are relatively consistent. More complex claim types such as liability, medical, or multi-party disputes remain beyond the present evaluation and would require significant domain adaptation. OCR performance is strong for printed text but still challenged by handwriting, non-Latin scripts, and documents with dense tabular or highly formatted layouts. The implementation targets Italian and English contexts, meaning broader international deployment would necessitate localized models, language handling, and regulatory adaptation.

System reliability and economics remain closely tied to external technology dependencies, particularly the OpenAI API, whose model pricing, throughput limits, and latency variability directly influence both performance and cost predictability. Sustained production use would benefit from model diversification, with fallbacks or fine-tuned local models mitigating risks of dependency, pricing changes, or service disruption while allowing continuous optimization of the accuracy–cost balance.

### 4.5.3 Practical Deployment Considerations

Automation delivers its greatest benefit in high-volume, routine claims where documentation is complete and policy interpretation follows predictable patterns. In these cases, automated workflows can process claims efficiently, applying consistent logic and producing verifiable results with minimal human intervention. The system is particularly well suited for low to medium value claims, for instance those below five thousand euros, and for coverage types governed by clear contractual terms. Cases involving high claim values, ambiguous or contradictory policy language, incomplete documentation, or scenarios near coverage boundaries remain better suited for human review, allowing expert oversight in decisions that carry greater financial or regulatory risk.

From an economic perspective, the system presents a favorable cost–benefit balance once claim volumes reach several thousand per month. Fixed investments in infrastructure, embeddings, and development amortize quickly across large workloads, while per-claim variable costs tied to LLM usage remain manageable through selective model invocation and caching. A complete break-even analysis should account for labor savings, improvements in customer satisfaction resulting from faster turnaround times, and reductions in error rates due to deterministic validation and consistent application of policy rules.

Deployment in production environments requires integration with existing claims management systems, customer databases, and policy repositories. The modular API-based architecture simplifies this integration, allowing the system to act as an intelligent processing layer between front-end interfaces and legacy systems. Rigorous data validation at integration points ensures consistency and prevents propagation of erroneous inputs. Built-in audit logging and explainability features reinforce regulatory compliance, enabling transparent review of each automated decision and supporting internal quality assurance procedures.

## 4.6 Chapter Summary

This chapter evaluated the AI-powered insurance claims processing system across document processing accuracy, claims validation performance, RAG-based policy interpretation, and overall system efficiency using synthetic datasets.

Results show that field extraction performs with near-perfect accuracy on digital PDFs and acceptable accuracy on scanned documents, while smartphone images remain the most error-prone due to lighting, skew, and resolution issues. The hybrid PDF and OCR pipeline, supported by intelligent document-type routing, significantly improves robustness and minimizes unnecessary computation. Claims validation achieves consistent accuracy on routine cases, combining the flexibility of LLM-based extraction with the determinism of rule-based validation and exact decimal arithmetic. Most residual errors arise from OCR inaccuracies rather than logic failures, indicating that upstream extraction quality remains the main factor influencing end-to-end reliability.

RAG-based policy interpretation demonstrates strong practical viability for customer queries, successfully grounding responses in retrieved clauses and reducing hallucination risk through explicit citation enforcement. The dual vectorstore design achieves full privacy compliance without compromising retrieval speed or accuracy. Query latency and cost remain well within production thresholds, confirming scalability for real-world deployment.

Overall system efficiency meets interactive expectations, with end-to-end processing times suitable for customer-facing applications and costs scaling predictably when LLM usage is optimized. The stateless API design enables straightforward horizontal scaling across multiple workers, supporting future expansion for higher claim volumes.

The evaluation's scope, limited to synthetic datasets and straightforward property and automotive claims, restricts statistical generalization and omits complex domains such as liability and medical claims. Nevertheless, the results provide a clear foundation for production deployment under a confidence-based routing strategy, automating standard cases while routing uncertain or high-value claims for human review. Continuous monitoring with real claims data will be essential to validate performance, refine thresholds, and ensure sustained compliance. These findings confirm that AI-driven claims automation can substantially enhance efficiency and consistency in insurance operations when combined with human oversight, demonstrating that hybrid human-AI workflows can meet both operational and regulatory standards.

# Chapter 5

# Conclusions

## 5.1 Background and Motivation

## 5.2 Summary of Contributions

This thesis explored the automation of insurance claims assessment through the integration of Large Language Models, Retrieval-Augmented Generation, and advanced OCR techniques, demonstrating that intelligent automation can transform a traditionally manual and time-consuming process into a streamlined, data-driven workflow. The research shows that AI-powered systems can deliver significant gains in processing speed and cost efficiency while maintaining the high standards of accuracy, interpretability, and regulatory compliance required in the insurance sector. By combining scalable machine intelligence with transparent reasoning and human oversight mechanisms, the implemented architecture illustrates how next-generation insurance platforms can achieve operational excellence without sacrificing trust, accountability, or customer confidence.

### 5.2.1 Technical Contributions

The central technical contribution of this thesis lies in the design and implementation of a production-grade system architecture that integrates multiple AI components, retrieval-augmented generation, optical character recognition, and large language model orchestration, into a unified and regulatory-compliant workflow for insurance claims processing. Each subsystem embodies a design pattern that addresses a specific technical and compliance challenge while remaining generalizable to other data-intensive and privacy-sensitive domains.

The dual vectorstore architecture represents a foundational advance in the application of RAG within regulated environments. By maintaining strict separation between permanent and ephemeral knowledge stores, it resolves the tension between retrieval performance and data protection. Permanent corporate documents such as policy texts and contractual terms reside indefinitely in a persistent ChromaDB instance optimized for large-scale semantic retrieval, while customer-uploaded materials are stored in a temporary vectorstore created per session and automatically deleted upon cleanup or system shutdown. This design enforces GDPR's principles of storage limitation and data minimization at the architectural level, ensuring that personal data never becomes entangled with permanent knowledge sources. Beyond insurance, this separation model offers a reusable blueprint for sectors such as finance, healthcare, and legal services, where retrieval systems must balance persistent institutional memory with transient and privacy-constrained user data.

The hybrid OCR processing pipeline advances document understanding by tailoring processing to document characteristics rather than relying on a uniform approach. Digital PDFs are routed through native text extractors such as PyMuPDF and PDFMiner, providing near-instant results with perfect fidelity when structured text layers are available. Scanned or degraded documents are automatically detected and processed through a six-stage enhancement pipeline consisting of grayscale normalization, deskewing, contrast-limited adaptive histogram equalization, denoising, adaptive thresholding, and morphological refinement before being interpreted via Tesseract OCR. This adaptive routing minimizes unnecessary computational cost and maximizes recognition accuracy across heterogeneous document types. The result is a cost-efficient and high-accuracy OCR subsystem capable of handling real-world insurance artifacts ranging from machine-generated policies to low-quality mobile photographs, all within a unified abstraction accessible to upstream services.

The tiered LLM selection strategy contributes an empirically validated framework for optimizing quality and cost balance in multi-model production systems. Lightweight models such as GPT-3.5-turbo are allocated to deterministic and low-context tasks including identifier extraction, coverage classification, and structured field parsing, while GPT-4 is reserved for tasks requiring deep contextual reasoning such as interpreting multi-clause policy wording or evaluating borderline claim scenarios. Benchmarking shows that this task-based tiering achieves comparable decision quality to all-GPT-4 configurations while reducing token costs by over half, providing a scalable operational template for any enterprise deploying LLMs under budgetary constraints. This cost-aware orchestration strategy demonstrates how careful task decomposition can preserve reasoning fidelity while maintaining economic feasibility.

Finally, the grounded generation mechanisms provide a structural safeguard against hallucination, a critical requirement in compliance-bound sectors. By embedding citation prompts into RAG queries and validating that generated responses reference retrieved contract clauses, the system ensures factual grounding of every answer. The resulting AI behavior remains transparent and verifiable, as each response can be traced back to the precise contractual provisions that informed it. This explicit grounding not only prevents the dissemination of fabricated or non-contractual information but also satisfies explainability and accountability requirements under frameworks such as the Insurance Distribution Directive and the EU AI Act. In doing so, it bridges the operational advantages of AI-driven automation with the evidentiary rigor demanded by regulatory oversight.

Together, these innovations, privacy-preserving vectorstore separation, adaptive OCR orchestration, model-tiered reasoning, and citation-grounded generation, constitute a cohesive architecture that demonstrates how advanced AI technologies can be responsibly deployed in safety-critical and legally regulated domains.

### 5.2.2 Methodological Contributions

The synthetic data generation methodology establishes a privacy-compliant foundation for evaluating AI systems in data-sensitive environments. By using GPT-4 to generate realistic yet artificial insurance datasets that include customer records, policy contracts, and claim scenarios, structured through carefully designed prompts with built-in consistency validation, the approach allows end-to-end testing without exposing personal information. It enables rigorous experimentation, benchmarking, and public demonstration while adhering to privacy regulations that prohibit sharing real customer data. Although synthetic data lacks the full variability and subtlety of real-world datasets, it captures sufficient structural and semantic fidelity to meaningfully evaluate model reasoning, system integration, and compliance workflows, thus offering a viable substitute for sensitive proprietary data.

The evaluation framework redefines AI performance assessment for the insurance context by replacing general-purpose linguistic metrics with retrieval-augmented generation measures that reflect regulatory and operational priorities. Metrics such as faithfulness, citation accuracy, and hallucination rate directly evaluate the factual integrity and explainability of generated outputs, aligning model assessment with compliance standards that require verifiable and traceable decision logic. The evaluation spans multiple dimensions including OCR extraction accuracy, RAG faithfulness, overall system responsiveness, cost efficiency, and agreement between AI-generated and human-reviewed decisions, providing a holistic validation of both technical robustness and business readiness. This framework offers a replicable model for assessing AI systems in regulated domains where reliability, interpretability, and accountability are more critical than stylistic or lexical precision.

## 5.3 Limitations and Future Enhancements

While the system demonstrates technical feasibility and operational viability, several limitations suggest directions for future research and development.

### 5.3.1 Data and Evaluation Limitations

The most significant limitation is evaluation on synthetic rather than real data, since although synthetic data generation enabled privacy-compliant experimentation and comprehensive testing, such datasets inevitably fail to reproduce the full complexity and irregularity of real insurance claims. Actual claims often contain heterogeneous document structures, ambiguous or contradictory statements, incomplete information, and even adversarial content such as fraudulent or misleading submissions, all of which challenge AI systems in ways that curated synthetic data rarely does. To ensure robustness and external validity, future research should incorporate real insurance data obtained under controlled privacy agreements, enabling quantitative measurement of generalization from synthetic to authentic conditions and precise identification of failure modes that emerge only in production environments.

Human evaluation limitations further constrain confidence in RAG faithfulness and reliability assessments, as manual verification of faithfulness, citation accuracy, and hallucination rate remains time-consuming, subjective, and sensitive to annotator interpretation. The limited scale of human review restricts statistical confidence and reproducibility, making it difficult to detect subtle degradations across updates or model versions. Future work should develop automated or semi-automated faithfulness metrics based on natural language inference and factual consistency modeling to approximate human judgments at scale, enabling continuous, low-cost monitoring of model integrity. Moreover, the current evaluation framework provides only a static snapshot of performance; it does not measure longitudinal stability as policies evolve, new claim categories appear, or underlying LLM APIs change. Implementing continuous evaluation pipelines that periodically re-assess accuracy, grounding quality, and system cost-efficiency would ensure sustained compliance and performance over time, a prerequisite for dependable production deployment.

### 5.3.2 Functional Enhancements

Client identification could be extended beyond current capabilities to support broader input formats and ambiguity resolution. While the system currently handles explicit customer IDs and name-based matching, further improvements could enable identification through alternative identifiers such as policy numbers, partial names, or incomplete information. This could involve integrating lightweight Named Entity Recognition models fine-tuned on insurance-specific entities, or adopting few-shot prompting strategies where LLMs are guided with examples of valid identifiers. Such enhancements would allow the system to infer intent and identify customers even when information is indirect or incomplete, aligning more closely with natural conversational behavior.

Unified cross-vectorstore retrieval would enable seamless reasoning across permanent and temporary knowledge sources. At present, policy-related queries access only the permanent vectorstore while uploaded documents are handled separately. Implementing joint retrieval that simultaneously queries both sources, merges results according to normalized similarity scores, and removes duplicates would allow cross-referenced analysis such as verifying whether an uploaded receipt aligns with a policy's deductible clause. This enhancement requires intelligent result fusion and context window optimization to prevent prompt overloading while maintaining balanced coverage between persistent and transient content.

Multi-modal evidence processing presents another direction for advancement. The current OCR pipeline processes textual information effectively, yet insurance claims frequently include images of vehicle damage, scanned signatures, or diagrams that require visual understanding. Incorporating multi-modal models such as GPT-4 Vision would enable direct reasoning over visual content, supporting use cases like verifying whether photographed damage corresponds to the claim description or whether a signature matches stored templates. Given the high computational and financial cost of visual models, future versions could apply selective activation, invoking vision-based analysis only when image metadata or context indicates potential relevance.

Session-based conversation management would improve continuity and efficiency in user interaction. Each current request is stateless, requiring users to repeat identifiers or reintroduce context for follow-up questions. Introducing session persistence, either through browser-side local storage or server-managed session databases with expiration policies, would allow the system to remember recent clients, coverage types, and retrieved documents. This enhancement would enable fluid multi-turn dialogues where users can naturally refine or extend previous queries without re-specification, creating a more conversational and user-friendly experience while preserving privacy through scoped, time-limited session memory.

### 5.3.3 Production Readiness Enhancements

Role-based access control (RBAC) is essential to enforce least-privilege principles and safeguard sensitive data. The current architecture allows unrestricted access to all API endpoints, which is acceptable for internal or prototype deployments but insufficient for production environments handling real customer data. Introducing RBAC would restrict endpoint access according to user roles, ensuring that customers can query only their own policies, adjusters can view claims within their jurisdiction, and administrators can modify configuration or monitor system health. This can be achieved through OAuth 2.0 or comparable authentication frameworks, integrating identity providers for token-based verification and augmenting database queries with row-level security, ensuring alignment with enterprise-grade security standards and regulatory expectations for controlled data access.

Rate limiting and abuse protection are critical to maintaining operational stability and cost predictability. In the absence of rate limits, a malfunctioning client or intentional misuse could trigger excessive queries, exhausting API quotas or generating unbounded expenses. Implementing tiered limits based on user roles, for example ten queries per minute for customers, one hundred for adjusters, and unrestricted access for administrators, would preserve fair resource allocation while preventing denial of service conditions. Graceful degradation mechanisms, such as temporary queuing rather than abrupt rejection near thresholds, would maintain service continuity while discouraging overuse.

Comprehensive audit logging strengthens regulatory compliance and operational transparency. Extending the existing structured logging system into a full audit trail would record user identities, action categories, timestamps, data accessed, and decision outcomes in immutable, tamper-evident storage. Such logs would support compliance audits by demonstrating human oversight in automated decisions, enable forensic analysis of security incidents, and facilitate continuous quality assurance by detecting recurring model errors or anomalous user behavior. Audit record retention and access policies would need to follow GDPR and IDD constraints, balancing accountability with privacy.

Distributed vectorstore deployment represents a key scalability enhancement. The embedded configuration of ChromaDB limits concurrency and dataset size to a single host, constraining throughput for large-scale insurance operations. Migrating to a distributed configuration, either through ChromaDB's client-server mode or managed vector database platforms such as Pinecone, Weaviate, or Qdrant, would allow the system to handle millions of document chunks and thousands of simultaneous queries. This transition introduces infrastructure complexity, requiring network orchestration, replica synchronization, and observability, but enables horizontal scaling and fault-tolerant retrieval suitable for enterprise-grade deployments.

## 5.4 Concluding Remarks

The integration of Large Language Models, Retrieval-Augmented Generation, and advanced OCR processing represents a major advancement in insurance claims automation, showing that modern AI can effectively support human decision-making in complex regulated domains while preserving the transparency, accountability, and human oversight demanded by law and customer trust. This thesis demonstrates that AI-powered claims processing is not only technically feasible but also operationally viable, achieving interactive response times, economically sustainable costs, and accuracy levels comparable to human adjusters in routine cases.

The architectural patterns developed in this research, including dual vectorstore separation, hybrid document processing, tiered LLM selection, and grounded generation with explicit citations, extend beyond the insurance industry and can be applied to other regulated sectors facing similar challenges. Healthcare organizations managing medical records and knowledge repositories, financial institutions processing loan applications under strict regulatory frameworks, legal professionals interpreting case law and statutes, and government agencies administering benefits programs all encounter comparable requirements. They must combine large, persistent knowledge bases with sensitive, short-lived user data, handle diverse document formats, perform complex reasoning grounded in authoritative sources, and comply with stringent transparency and auditability standards. The system architecture and evaluation methodology presented in this thesis provide a transferable blueprint for deploying AI solutions that satisfy these cross-domain demands.

The human-in-the-loop design philosophy underpinning the system embodies a pragmatic vision of AI deployment in high-stakes environments. Instead of seeking full automation, the system uses AI to resolve routine cases representing roughly half of all claims while escalating ambiguous or low-confidence cases to human experts. This balanced approach maximizes efficiency by automating predictable tasks while retaining expert oversight where judgment and context are critical. As model performance and retrieval precision improve, the boundary between automated and human-reviewed cases will continue to evolve, yet the guiding principle of AI augmentation rather than replacement will remain essential in sectors where accountability and trust define legitimacy.

Future progress will emerge from the convergence of increasingly capable foundation models, enhanced retrieval and grounding mechanisms, and richer domain-specific training datasets. However, success will depend as much on governance as on technology. Effective AI deployment in regulated environments demands attention to compliance, privacy, explainability, and oversight from the earliest design stages. This thesis illustrates that these constraints, far from limiting innovation, can shape architectures that are more resilient, transparent, and aligned with institutional and societal expectations. Sustainable AI in regulated domains is achieved not by circumventing regulation but by internalizing it, turning compliance into a structural principle that strengthens both technical integrity and public confidence.

# Bibliography

[1] Frederick Hayes-Roth, D. A. Waterman, and Douglas B. Lenat. *Building Expert Systems*. Accessed: 2025-11-27. Addison-Wesley Pub. Co., 1983. ISBN: 0201106868. URL: `https://archive.org/details/buildingexpertsy00temd` (cit. on p. 7).

[2] S. Mori, C.Y. Suen, and K. Yamamoto. «Historical review of OCR research and development». In: *Proceedings of the IEEE* 80.7 (1992), pp. 1029–1058. DOI: `10.1109/5.156468` (cit. on pp. 7, 28).

[3] Stijn Viaene, Richard A. Derrig, Bart Baesens, and Guido Dedene. «A Comparison of State-of-the-Art Classification Techniques for Expert Automobile Insurance Claim Fraud Detection». In: *Journal of Risk and Insurance* 69.3 (2002), pp. 373–421. DOI: `10.1111/1539-6975.00023`. URL: `https://doi.org/10.1111/1539-6975.00023` (cit. on p. 7).

[4] Sharmila Subudhi and Suvasini Panigrahi. «Use of optimized Fuzzy C-Means clustering and supervised classifiers for automobile insurance fraud detection». In: *Journal of King Saud University - Computer and Information Sciences* 32.5 (2020), pp. 568–575. ISSN: 1319-1578. DOI: `https://doi.org/10.1016/j.jksuci.2017.09.010`. URL: `https://www.sciencedirect.com/science/article/pii/S1319157817301672` (cit. on p. 7).

[5] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: `10.1109/5.726791` (cit. on p. 8).

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: `1512.03385 [cs.CV]`. URL: `https://arxiv.org/abs/1512.03385` (cit. on p. 8).

[7] R. Smith. «An Overview of the Tesseract OCR Engine». In: *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*. Vol. 2. 2007, pp. 629–633. DOI: `10.1109/ICDAR.2007.4376991` (cit. on pp. 8, 28).

[8] Alex Graves, Marcus Liwicki, Santiago Fernández, Ramon Bertolami, Horst Bunke, and Jürgen Schmidhuber. «A novel connectionist system for unconstrained handwriting recognition». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31.5 (2009), pp. 855–868. DOI: `10.1109/TPAMI.2008.137`. URL: `https://doi.org/10.1109/TPAMI.2008.137` (cit. on p. 8).

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: `1706.03762 [cs.CL]`. URL: `https://arxiv.org/abs/1706.03762` (cit. on p. 8).

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: `1810.04805 [cs.CL]`. URL: `https://arxiv.org/abs/1810.04805` (cit. on pp. 8, 40).

[11] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: `2005.14165 [cs.CL]`. URL: `https://arxiv.org/abs/2005.14165` (cit. on pp. 8, 10–12).

[12] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: `2303.08774 [cs.CL]`. URL: `https://arxiv.org/abs/2303.08774` (cit. on p. 8).

[13] Jason Wei et al. *Emergent Abilities of Large Language Models*. 2022. arXiv: `2206.07682 [cs.CL]`. URL: `https://arxiv.org/abs/2206.07682` (cit. on pp. 8, 12, 15).

[14] Michael Bommarito II and Daniel Martin Katz. *GPT Takes the Bar Exam*. 2022. arXiv: `2212.14402 [cs.CL]`. URL: `https://arxiv.org/abs/2212.14402` (cit. on p. 9).

[15] Daniel M. Katz, Michael J. Bommarito, Shang Gao, and Pablo Arredondo. «GPT4 passes the bar exam». In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 382.2246 (2024). DOI: `10.1098/rsta.2023.0179`. URL: `https://doi.org/10.1098/rsta.2023.0179` (cit. on p. 9).

[16] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2021. arXiv: `2005.11401 [cs.CL]`. URL: `https://arxiv.org/abs/2005.11401` (cit. on pp. 9, 24).

[17] Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. *Precise Zero-Shot Dense Retrieval without Relevance Labels*. 2022. arXiv: `2212.10496 [cs.IR]`. URL: `https://arxiv.org/abs/2212.10496` (cit. on pp. 9, 24).

[18] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. «Learning representations by back-propagating errors». In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0 (cit. on p. 10).

[19] Sepp Hochreiter and Jürgen Schmidhuber. «Long Short-Term Memory». In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735 (cit. on p. 10).

[20] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. *Generating Long Sequences with Sparse Transformers.* 2019. arXiv: 1904.10509 [cs.LG]. URL: https://arxiv.org/abs/1904.10509 (cit. on p. 10).

[21] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. *Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention.* 2020. arXiv: 2006.16236 [cs.LG]. URL: https://arxiv.org/abs/2006.16236 (cit. on p. 10).

[22] Dan Hendrycks and Kevin Gimpel. *Gaussian Error Linear Units (GELUs).* 2023. arXiv: 1606.08415 [cs.LG]. URL: https://arxiv.org/abs/1606.08415 (cit. on p. 10).

[23] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. *Self-Attention with Relative Position Representations.* 2018. arXiv: 1803.02155 [cs.CL]. URL: https://arxiv.org/abs/1803.02155 (cit. on p. 11).

[24] Ruibin Xiong et al. *On Layer Normalization in the Transformer Architecture.* 2020. arXiv: 2002.04745 [cs.LG]. URL: https://arxiv.org/abs/2002.04745 (cit. on p. 11).

[25] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. «Improving Language Understanding by Generative Pre-Training». In: (2018). preprint. URL: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf (cit. on p. 11).

[26] Jeremy Howard and Sebastian Ruder. *Universal Language Model Fine-tuning for Text Classification.* 2018. arXiv: 1801.06146 [cs.CL]. URL: https://arxiv.org/abs/1801.06146 (cit. on p. 11).

[27] Rico Sennrich, Barry Haddow, and Alexandra Birch. «Neural Machine Translation of Rare Words with Subword Units». In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* Ed. by Katrin Erk and Noah A. Smith. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. DOI: 10.18653/v1/P16-1162. URL: https://aclanthology.org/P16-1162/ (cit. on p. 11).

[28] Taku Kudo and John Richardson. *SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing.* 2018. arXiv: 1808.06226 [cs.CL]. URL: https://arxiv.org/abs/1808.06226 (cit. on p. 11).

[29] James Kirkpatrick et al. «Overcoming catastrophic forgetting in neural networks». In: *Proceedings of the National Academy of Sciences* 114.13 (Mar. 2017), pp. 3521–3526. ISSN: 1091-6490. DOI: 10.1073/pnas.1611835114. URL: http://dx.doi.org/10.1073/pnas.1611835114 (cit. on p. 12).

[30] Jared Kaplan et al. *Scaling Laws for Neural Language Models.* 2020. arXiv: 2001.08361 [cs.LG]. URL: https://arxiv.org/abs/2001.08361 (cit. on p. 12).

[31] Paul Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. *Deep reinforcement learning from human preferences.* 2023. arXiv: 1706.03741 [stat.ML]. URL: https://arxiv.org/abs/1706.03741 (cit. on p. 13).

[32] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal Policy Optimization Algorithms.* 2017. arXiv: 1707.06347 [cs.LG]. URL: https://arxiv.org/abs/1707.06347 (cit. on p. 13).

[33] Ziwei Ji et al. «Survey of Hallucination in Natural Language Generation». In: *ACM Computing Surveys* 55.12 (Mar. 2023), pp. 1–38. ISSN: 1557-7341. DOI: 10.1145/3571730. URL: http://dx.doi.org/10.1145/3571730 (cit. on p. 15).

[34] Joshua Maynez, Shashi Narayan, Bernd Bohnet, and Ryan McDonald. «On Faithfulness and Factuality in Abstractive Summarization». In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics.* Ed. by Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault. Online: Association for Computational Linguistics, July 2020, pp. 1906–1919. DOI: 10.18653/v1/2020.acl-main.173. URL: https://aclanthology.org/2020.acl-main.173/ (cit. on p. 15).

[35] Emily M. Bender and Alexander Koller. «Climbing towards NLU: On Meaning, Form, and Understanding in the Age of Data». In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics.* Ed. by Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault. Online: Association for Computational Linguistics, July 2020, pp. 5185–5198. DOI: 10.18653/v1/2020.acl-main.463. URL: https://aclanthology.org/2020.acl-main.463/ (cit. on p. 16).

[36] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. «On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?» In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency (FAccT '21)*. 2021, pp. 610–623. DOI: 10.1145/3442188.3445922. URL: https://dl.acm.org/doi/10.1145/3442188.3445922 (cit. on p. 16).

[37] Tolga Bolukbasi, Kai-Wei Chang, James Zou, Venkatesh Saligrama, and Adam Kalai. *Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings*. 2016. arXiv: 1607.06520 [cs.CL]. URL: https://arxiv.org/abs/1607.06520 (cit. on p. 16).

[38] Zachary C. Lipton. *The Mythos of Model Interpretability*. 2017. arXiv: 1606.03490 [cs.LG]. URL: https://arxiv.org/abs/1606.03490 (cit. on p. 17).

[39] Gautier Izacard and Edouard Grave. *Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering*. 2021. arXiv: 2007.01282 [cs.CL]. URL: https://arxiv.org/abs/2007.01282 (cit. on p. 17).

[40] Omar Khattab and Matei Zaharia. *ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT*. 2020. arXiv: 2004.12832 [cs.IR]. URL: https://arxiv.org/abs/2004.12832 (cit. on p. 17).

[41] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. *Dense Passage Retrieval for Open-Domain Question Answering*. 2020. arXiv: 2004.04906 [cs.CL]. URL: https://arxiv.org/abs/2004.04906 (cit. on p. 19).

[42] Stephen E. Robertson and Hugo Zaragoza. «The Probabilistic Relevance Framework: BM25 and Beyond». In: *Foundations and Trends® in Information Retrieval* 3.4 (2009), pp. 333–389. DOI: 10.1561/1500000019. URL: https://doi.org/10.1561/1500000019 (cit. on p. 19).

[43] Nils Reimers and Iryna Gurevych. «Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks». In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Ed. by Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 3982–3992. DOI: 10.18653/v1/D19-1410. URL: https://aclanthology.org/D19-1410/ (cit. on pp. 19, 40).

[44] Arvind Neelakantan et al. *Text and Code Embeddings by Contrastive Pre-Training*. 2022. arXiv: 2201.10005 [cs.CL]. URL: https://arxiv.org/abs/2201.10005 (cit. on pp. 20, 41).

[45] Gee Y. Lee, Scott Manski, and Tapabrata Maiti. «Actuarial Applications Of Word Embedding Models». In: *ASTIN Bulletin* 50.1 (2020), pp. 1–24. DOI: 10.1017/AST.2019.19 (cit. on p. 20).

[46] Yu. A. Malkov and D. A. Yashunin. *Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs*. 2018. arXiv: 1603.09320 [cs.DS]. URL: https://arxiv.org/abs/1603.09320 (cit. on pp. 20, 42).

[47] Jeff Johnson, Matthijs Douze, and Hervé Jégou. *Billion-scale similarity search with GPUs*. 2017. arXiv: 1702.08734 [cs.CV]. URL: https://arxiv.org/abs/1702.08734 (cit. on p. 21).

[48] Herve Jégou, Matthijs Douze, and Cordelia Schmid. «Product Quantization for Nearest Neighbor Search». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.1 (2011), pp. 117–128. DOI: 10.1109/TPAMI.2010.57 (cit. on p. 21).

[49] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. *Lost in the Middle: How Language Models Use Long Contexts*. 2023. arXiv: 2307.03172 [cs.CL]. URL: https://arxiv.org/abs/2307.03172 (cit. on p. 26).

[50] Rodrigo Nogueira and Kyunghyun Cho. *Passage Re-ranking with BERT*. 2020. arXiv: 1901.04085 [cs.IR]. URL: https://arxiv.org/abs/1901.04085 (cit. on p. 27).

[51] Anthony Kay. «Tesseract: an Open-Source Optical Character Recognition Engine». In: *Linux Journal* 2007.155 (2007). Accessed: 2025-11-27. URL: https://dl.acm.org/doi/fullHtml/10.5555/1288165.1288167 (cit. on p. 28).

[52] Baoguang Shi, Xiang Bai, and Cong Yao. *An End-to-End Trainable Neural Network for Image-based Sequence Recognition and Its Application to Scene Text Recognition*. 2015. arXiv: 1507.05717 [cs.CV]. URL: https://arxiv.org/abs/1507.05717 (cit. on p. 29).

[53] Yiheng Xu, Minghao Li, Lei Cui, Shaohan Huang, Furu Wei, and Ming Zhou. «LayoutLM: Pre-training of Text and Layout for Document Image Understanding». In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*. KDD ’20. ACM, Aug. 2020, pp. 1192–1200. DOI: 10.1145/3394486.3403172. URL: http://dx.doi.org/10.1145/3394486.3403172 (cit. on p. 29).

[54] Yupan Huang, Tengchao Lv, Lei Cui, Yutong Lu, and Furu Wei. *LayoutLMv3: Pre-training for Document AI with Unified Text and Image Masking*. 2022. arXiv: 2204.08387 [cs.CL]. URL: https://arxiv.org/abs/2204.08387 (cit. on p. 29).

[55] Geewook Kim et al. *OCR-free Document Understanding Transformer.* 2022. arXiv: `2111.15664 [cs.LG]`. URL: `https://arxiv.org/abs/2111.15664` (cit. on p. 29).

[56] Nobuyuki Otsu. «A Threshold Selection Method from Gray-Level Histograms». In: *IEEE Transactions on Systems, Man, and Cybernetics* 9.1 (1979), pp. 62–66. DOI: `10.1109/TSMC.1979.4310076` (cit. on p. 32).

[57] Jaakko Sauvola and Mika Pietikäinen. «Adaptive document image binarization». In: *Pattern Recognition* 33.2 (2000), pp. 225–236. DOI: `10.1016/S0031-3203(99)00055-2`. URL: `https://www.sciencedirect.com/science/article/pii/S0031320399000552` (cit. on p. 32).

[58] C. Tomasi and R. Manduchi. «Bilateral filtering for gray and color images». In: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271).* 1998, pp. 839–846. DOI: `10.1109/ICCV.1998.710815` (cit. on p. 32).

[59] D. S. Le, G. R. Thoma, and H. Wechsler. «Automated page orientation and skew angle detection for binary document images». In: *Pattern Recognition* 27.10 (1994), pp. 1325–1344 (cit. on p. 32).

[60] M.S. Brown and W.B. Seales. «Document restoration using 3D shape: a general deskewing algorithm for arbitrarily warped documents». In: *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001.* Vol. 2. 2001, 367–374 vol.2. DOI: `10.1109/ICCV.2001.937649` (cit. on p. 33).

[61] Ram Krishna Pandey and A G Ramakrishnan. *Language Independent Single Document Image Super-Resolution using CNN for improved recognition.* 2017. arXiv: `1701.08835 [cs.CV]`. URL: `https://arxiv.org/abs/1701.08835` (cit. on p. 33).

[62] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. *Efficient Estimation of Word Representations in Vector Space.* 2013. arXiv: `1301.3781 [cs.CL]`. URL: `https://arxiv.org/abs/1301.3781` (cit. on p. 40).

[63] Jeffrey Pennington, Richard Socher, and Christopher Manning. «GloVe: Global Vectors for Word Representation». In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).* Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. DOI: `10.3115/v1/D14-1162`. URL: `https://aclanthology.org/D14-1162/` (cit. on p. 40).

[64] Harrison Chase. *LangChain: Building applications with LLMs through composability.* `https://github.com/langchain-ai/langchain`. Accessed: 2025-11-27. 2022 (cit. on p. 45).

[65] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. *"Why Should I Trust You?": Explaining the Predictions of Any Classifier*. 2016. arXiv: 1602.04938 [cs.LG]. URL: https://arxiv.org/abs/1602.04938 (cit. on p. 54).

[66] Tim Miller. «Explanation in artificial intelligence: Insights from the social sciences». In: *Artificial Intelligence* 267 (2019), pp. 1–38. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2018.07.007. URL: https://www.sciencedirect.com/science/article/pii/S0004370218305988 (cit. on p. 54).

[67] Nelson Cowan. «The magical number 4 in short-term memory: A reconsideration of mental storage capacity». In: *Behavioral and Brain Sciences* 24.1 (2001), pp. 87–185. DOI: 10.1017/S0140525X01003922. URL: https://doi.org/10.1017/S0140525X01003922 (cit. on p. 55).

[68] David B. Kaber and Mica R. Endsley. «The effects of level of automation and adaptive automation on human performance, situation awareness and workload in a dynamic control task». In: *Theoretical Issues in Ergonomics Science* 5.2 (2004), pp. 113–153. DOI: 10.1080/1463922021000054335. URL: https://doi.org/10.1080/1463922021000054335 (cit. on p. 55).

[69] Raja Parasuraman and Victor Riley. «Humans and Automation: Use, Misuse, Disuse, Abuse». In: *Human Factors* 39.2 (1997), pp. 230–253. DOI: 10.1518/001872097778543886. URL: https://doi.org/10.1518/001872097778543886 (cit. on p. 55).