

Vrije Universiteit Amsterdam



Universiteit van Amsterdam



Master Thesis

---

# Blueprint for Institutional LLM Adoption: On-Premise, Open-Source, and Domain-Aware

---

**Author:** Victor Wie (2828546)

*Supervisor:* Adam Belloum

*2nd reader:* Marios Avgeris

*A thesis submitted in fulfillment of the requirements for  
the joint UvA-VU Master of Science degree in Computer Science*

July 1, 2025

## Abstract

Large Language Models (LLMs) like ChatGPT are widely adopted across sectors, but their often closed-source and externally hosted nature poses challenges for data sovereignty, transparency, and compliance, especially in settings involving sensitive or domain-specific information. This thesis explores the viability of open-source LLMs deployed entirely on-premise as alternatives to commercial, cloud-based systems. It surveys the current landscape of open-source models, outlines their suitability for on-premise deployment, and examines methods for domain adaptation including prompt engineering, retrieval-augmented generation (RAG), and fine-tuning. To operationalize these ideas, a fully local RAG-based assistant was implemented and evaluated using structured university policy documents. The system runs without internet access, using a compact open-source model for inference, local embedding for semantic retrieval, and a modular interface for constrained document-grounded generation. Evaluation across two hardware profiles demonstrates high retrieval accuracy, low semantic error rates, and acceptable latency under resource-constrained conditions. The case study confirms that domain-specific, private LLM deployment is technically feasible on consumer hardware and offers practical advantages in transparency, data control, and adaptability.

# Contents

<b>List of Figures</b>	<b>iii</b>
<b>List of Tables</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation and Context . . . . .	1
1.2 Problem Statement and Research Gap . . . . .	3
1.3 Research Questions (RQs) . . . . .	4
1.4 Approach and Contributions . . . . .	4
1.5 Thesis Structure . . . . .	5
<b>2 Background</b>	<b>9</b>
2.1 Introduction to Large Language Models . . . . .	9
2.1.1 LLM Families . . . . .	11
2.1.2 The Open vs. Closed-Source LLM Paradigm . . . . .	14
2.2 Methods for Domain Adaptation . . . . .	16
2.2.1 General Overview . . . . .	17
<b>3 Overview</b>	<b>23</b>
3.1 System Overview . . . . .	23
3.2 Use Case and User Interaction Flow . . . . .	24
3.2.1 Student Flow . . . . .	25
<b>4 Design</b>	<b>27</b>
4.1 Design Rationale and Goals . . . . .	27
4.2 System Architecture and Data Flow . . . . .	28
4.3 Component-Level Design Decisions . . . . .	31

## CONTENTS

---

<b>5</b>	<b>Implementation</b>	<b>35</b>
5.1	Overview and Codebase . . . . .	35
5.2	Data Ingestion and Preprocessing . . . . .	37
5.3	Retrieval and Generation Engine . . . . .	38
5.4	User Interface . . . . .	41
5.5	Evaluation Infrastructure . . . . .	45
<b>6</b>	<b>Evaluation</b>	<b>47</b>
6.1	Experimental Setup and Design . . . . .	48
6.2	Evaluation Results . . . . .	49
6.3	Summary and Findings . . . . .	55
<b>7</b>	<b>Discussion</b>	<b>57</b>
7.1	<b>RQ1.1:</b> Landscape of Open-Source LLMs and Tools . . . . .	57
7.2	<b>RQ1.2:</b> Domain Adaptation and Integration . . . . .	58
7.3	<b>RQ1.3:</b> Comparative Viability of On-Premise Deployment . . . . .	58
7.4	Limitations and Future Work . . . . .	59
<b>8</b>	<b>Related Work</b>	<b>61</b>
8.1	Building UniGPT (Radas et al., 2023) . . . . .	61
8.2	Chat With OPD (Bhise, 2024) . . . . .	62
8.3	UvA AI Chat (Pepijn Stoop, 2025) . . . . .	63
8.4	On-Premises Knowledge Repository (Dobur et al., 2024) . . . . .	63
8.5	Towards On-Premise Hosted LLM (Hedlund, 2024) . . . . .	64
8.6	On-Site Deployment (Schillaci, 2024) . . . . .	64
8.7	MedAide (Basit et al., 2024) . . . . .	65
8.8	Summary and Alignment with Research Questions . . . . .	66
	8.8.1 Honorable Mentions and Implementation Influences . . . . .	66
	8.8.2 Research Gap and Contribution . . . . .	66
<b>9</b>	<b>Conclusion</b>	<b>69</b>
9.1	Summary of Approach and Findings . . . . .	69
9.2	Answering the Research Questions . . . . .	70
9.3	Outlook and Future Work . . . . .	70

# List of Figures

2.1	Retrieval and generation flow. Adapted from LangChain documentation [47].	18
2.2	LLM Agent Architecture. Courtesy of [48]. . . . .	20
3.1	(Prospective) Student User Flow . . . . .	26
4.1	High-level System Architecture . . . . .	29
5.1	Codebase layout with functional annotations. . . . .	36
5.2	Ingestion pipeline with four core stages. Adapted from LangChain documentation [47]. . . . .	37
5.3	Chatbot interface . . . . .	42
5.4	Automatically generated support inquiry based on chat history. . . . .	44

## LIST OF FIGURES

---

# List of Tables

2.1	Characteristics of Major LLM Families . . . . .	13
2.2	Comparison of open-source and closed-source LLMs . . . . .	16
4.1	Design goals and their implementation in the system architecture. . . . .	28
6.1	Pass rates desktop . . . . .	50
6.2	Average latency by topic and execution mode on desktop (n=30 per row). .	51
6.3	Average similarity scores by topic and execution mode (desktop). . . . .	52
6.4	Cross-system comparison of evaluation metrics (n=10 per laptop row, n=30 per desktop row). . . . .	54

## LIST OF TABLES

---



# Introduction

This chapter establishes the foundation for the thesis. It begins by outlining the broader context in which large language models are deployed and the institutional challenges they pose. It then defines the specific research gap this work addresses, formulates the guiding questions, and describes the approach taken. Finally, it summarizes the thesis structure, setting the stage for the technical and practical work that follows.

## 1.1 Motivation and Context

The rapid advancement of LLMs has driven their widespread adoption across industries, offering new opportunities for productivity, automation, and content generation. According to the 2024 Stack Overflow Developer Survey [1], approximately 75% of developers already use LLMs during code development. In parallel, recent market research estimates that by the end of 2025, over 750 million applications will rely on LLMs, with 50% of digital work expected to be automated through such tools [2]. Despite concerns around cost, accuracy, and data privacy, the global LLM market is projected to grow at a CAGR of nearly 80% through 2030, reflecting sustained momentum and enterprise interest in deploying these models operationally.

Despite their advantages, LLMs delivered through external, closed-source platforms raise critical concerns about transparency, auditability, and organizational control over data processing. This is particularly important for organizations that handle sensitive data that cannot be freely shared with external parties due to legal restrictions. Beyond the risks posed by closed-source models, the very act of transmitting data to third-party providers introduces undesirable exposure, creating vulnerabilities that could be avoided with local deployed. Although industry leaders like OpenAI state that user data is not used to train

## 1. INTRODUCTION

---

or improve their models unless users explicitly opt in, users must ultimately trust these assurances without the ability to verify them [3], [4]. Furthermore, regulations like the European Union’s AI Act [5], and other data protection frameworks like GDPR [6] and HIPAA [7], further underscore the need for transparent and accountable AI solutions [8].

In addition to legal and governance risks, large-scale commercial LLMs carry substantial environmental costs [9]. Recent analyses highlight that cloud-based AI systems, including popular services like ChatGPT, consume considerable energy and water per query due to centralized, GPU-intensive infrastructure and data center cooling requirements [10], [11]. Empirical studies estimate that training and serving foundation models like GPT-3 and GPT-4 involve tens to hundreds of megawatt-hours of electricity and generate nontrivial greenhouse gas emissions [12]. Moreover, the environmental footprint scales with both model size and the general-purpose scope of such systems, most of which are trained on broad, internet-scale corpora designed to support nearly universal knowledge domains. However, in institutional settings such as universities, assistants rarely require comprehensive coverage of, for example, culinary recipes or insect taxonomy. Deploying such overgeneralized models for constrained tasks introduces both unnecessary computational overhead and inflates energy use without corresponding benefit. From this perspective, task-specific, locally hosted models present a more sustainable alternative: they reduce emissions, minimize resource draw, and better align compute intensity with real-world application scope.

In sum, open-source alternatives enable internal deployment and oversight, reducing reliance on external infrastructure and limiting data exposure. Recent regulatory perspectives, such as those from the *European Insurance and Occupational Pensions Authority*, highlight the strategic role of open-source tooling in managing systemic risks, including those related to energy use and climate impact [13]. In well-scoped domains, local deployment avoids overprovisioning and aligns resource use with task requirements.

The challenge, then, is not merely to switch from commercial to open-source solutions, but to determine how organizations can deploy, customize, and govern these models internally without compromising data protection or operational effectiveness. While interest in on-premise LLMs is growing, existing research rarely offers end-to-end guidance on how to select, integrate, and adapt open-source models in institutional settings.

This thesis addresses these gaps by examining how organizations can bring LLM deployment in-house, retain full control over data flows, and increase efficiency in regulated contexts by allowing open-source models to access domain-specific data. For example, in a

university setting, a locally hosted and domain-adapted LLM could streamline administrative tasks, all while preserving the confidentiality of student and faculty data. Although this work draws on a university use case, the roadmap is intended to be broadly applicable: private individuals, government agencies, small-to-medium enterprises, and heavily regulated industries can similarly benefit from a well-defined strategy for secure, domain-relevant LLM deployments that uphold data sovereignty.

## 1.2 Problem Statement and Research Gap

Existing academic efforts in on-premise LLM deployment tend to address only parts of the broader challenge. Some works focus only on using RAG to increase domain-specific knowledge without allowing for reproducibility [14], while others demonstrate generic deployments of non-adapted models [15], [16]. Although each contribution provides great value, none delivers a single end-to-end framework that guides on-premise LLM adoption, from initial model and tool selection to domain-specific knowledge integration and data sovereignty. This thesis aims to close that gap by consolidating the practical steps needed for secure, high-performance, domain-relevant deployments.

Bhise et al. [14] propose a conceptual RAG pipeline for privacy-sensitive settings, but the system lacks source code, reproducible deployment steps, and empirical evaluation. It also omits model selection, architectural modularity, and generalization beyond the initial domain. Radas et al. [15] present UniGPT, a university-hosted assistant using open-source models, but focus mainly on infrastructure setup. While they briefly reference the potential for RAG, no retrieval pipeline is implemented, nor are the implications of document-grounded responses or domain adaptation discussed. Finally, the University of Amsterdam chatbot [16] relies on Microsoft’s Azure-hosted GPT models, explicitly sidestepping on-premise deployment. While pedagogically innovative, their system does not address transparency, reproducibility, or institutional control over model behavior and data exposure.

This thesis closes those gaps by delivering a fully self-contained, reproducible implementation that includes: (i) documented model and tool selection, (ii) a functioning document-grounded RAG pipeline tailored to a structured regulatory domain, and (iii) a modular architecture designed for local deployment without reliance on third-party APIs. It synthesizes retrieval, deployment, and privacy control into a coherent, end-to-end blueprint for domain-specific, locally hosted LLM applications.

## 1. INTRODUCTION

---

### 1.3 Research Questions (RQs)

To address the challenges outlined above, this thesis is guided by one main RQ:

*How can organizations adopt and adapt open-source LLMs on-premise to achieve data sovereignty and domain-specific performance, while overcoming the limitations of closed-source alternatives?*

In support of this overarching RQ, the work is structured around three sub-questions, each mapping to a distinct set of objectives and practical tasks:

**RQ1.1 Landscape of LLM Options:** *What open-source LLM models and supporting tools are available for on-premise deployment, and what are their characteristics in terms of architecture, model sizes and performance?*

**RQ1.2 Domain Adaptation:** *What methods are available for domain adaptation in on-premise LLMs, and how can internal documents be securely incorporated to improve domain-specific performance under hardware and privacy constraints?*

**RQ1.3 Comparative Analysis:** *Under what conditions do on-premise open-source LLM solutions offer strategic advantages over commercial LLM services, and how do model size and optimization strategies influence these trade-offs in domain-specific applications?*

### 1.4 Approach and Contributions

Each of the following points will describe their respective sub-RQ’s approach and contributions.

1. **RQ1.1:** To address this question, the thesis surveys the current landscape of open-source LLMs and supporting tools suitable for on-premise deployment. It identifies prominent model families and outlines their architectural characteristics, parameter scales, hardware requirements, and performance benchmarks.

The contribution is a structured, implementation-oriented overview of the open-source LLM ecosystem. It serves as a foundational reference for organizations exploring local deployments, offering guidance on models and tools suited to specific hardware, modularity, and offline inference constraints.

2. **RQ1.2:** To address this question, the thesis first surveys major methods for domain adaptation: prompt engineering, fine-tuning, and RAG, analyzing their technical requirements, privacy implications, and suitability for on-premise use. This conceptual analysis informs the system design, which implements a document-grounded RAG pipeline using static prompt templates and scoped retrieval over structured institutional texts. Prompt engineering is used to constrain model outputs to retrieved content, while short-term chat history supports follow-up queries. The contribution is twofold: a comparative review of adaptation strategies in the context of local deployment, and a practical demonstration of how RAG can be applied securely and effectively in a resource-constrained institutional setting.
3. **RQ1.3:** This question investigates the conditions under which on-premise open-source LLMs offer strategic advantages over commercial LLM services, with a focus on latency, retrieval accuracy, data sovereignty, and control over model behavior. The analysis draws on empirical results from the implemented system, including performance variation across hardware profiles, prompt sensitivity, and the impact of model scale on response quality and transparency.

The central hypothesis is that smaller, domain-adapted open-source models - when optimized for constrained environments - can match or exceed the practical effectiveness of commercial alternatives for well-scoped institutional tasks. This includes not only examining whether such models are viable, but when and why they should be preferred.

The contribution is a comparative framework based on real-world deployment scenarios, offering guidance on how trade-offs in auditability, privacy, and domain alignment influence the choice between local and externally hosted LLMs.

## 1.5 Thesis Structure

The chapters that follow reflect the steps taken in this work, beginning with the technical background and ending with an evaluation of the system and its broader implications.

- **Chapter 2 - Background:** Introduces the technical and conceptual foundations necessary for understanding the rest of the thesis. It outlines the core principles of LLMs, contrasts open-source and closed-source approaches, and reviews methods for domain adaptation.

## 1. INTRODUCTION

---

- **Chapter 3 - Overview:** Presents a high-level view of the proposed student desk chatbot system from the user’s perspective. It outlines the system’s purpose, architecture, key components, and intended interaction flow, establishing the context for later technical detail.
- **Chapter 4 - Design:** Explains the methodological decisions behind the system architecture and how they map to the research questions. It covers the rationale for model and tool selection, strategies for on-premise deployment, the approach to domain adaptation, and the evaluation design used to assess system behavior under realistic constraints.
- **Chapter 5 - Implementation:** Describes how the system was developed and assembled in practice. It includes details on local setups, document ingestion, RAG logic, interface behavior, and steps taken to maintain privacy, traceability, and modularity.
- **Chapter 6 - Evaluation:** Describes the experimental setup and presents empirical results from testing the system across multiple domains and runtime conditions. It evaluates the pipeline’s retrieval fidelity, generation correctness, and latency under both cold and warm execution modes on two hardware profiles. The chapter also analyzes failure cases, the role of prompt design and chunking in retrieval precision, and reflects on the effectiveness of an automated LLM-based judge for semantic evaluation.
- **Chapter 7 - Discussion:** Analyzes how the implemented system answers each research question, based on empirical results. It discusses the suitability of open-source models for local deployment (**RQ1.1**), the effectiveness and limitations of RAG-based domain adaptation (**RQ1.2**), and the trade-offs between open and closed LLMs in practical settings (**RQ1.3**). The chapter also reflects on system limitations and outlines opportunities for future refinement.
- **Chapter 8 - Related Work:** Reviews scientific and applied efforts in local LLM deployment, with a focus on RAG pipelines, privacy-preserving inference, and institutional use cases. It identifies limitations in scope, reproducibility, and evaluation across existing systems, and clarifies how this thesis contributes a fully implemented, domain-grounded, and benchmarked alternative.

- **Chapter 9 - Conclusion:** Synthesizes the thesis findings and answers the research questions in light of the system design and evaluation. It restates the viability of on-premise, open-source LLMs for domain-specific tasks, and proposes directions for extending the architecture through fine-tuning, real-world testing, and broader institutional deployment.

## 1. INTRODUCTION

---



## 2

# Background

This chapter is meant to set a common ground of understanding when it comes to the main topics discussed in this thesis. Firstly, we will provide a brief introduction to LLMs and how they work, before discussing the two paradigms, open vs. closed-source LLMs. Further, we define a focus and cover three foundational LLM families, before providing a taxonomy of domain adaptation methods.

## 2.1 Introduction to Large Language Models

Language models (LMs) include both statistical and neural models trained to predict the likelihood of a sequence of words, making it possible to generate coherent and contextually appropriate text [17]. Traditional LMs, such as n-gram models, rely on fixed-length contexts and struggle with capturing long-range dependencies or semantic structure [18]. The development of *large* language models was made possible by advances in neural architectures, most notably the *transformer*, which supports scalable capacity and data-driven learning in natural language tasks. LLMs like GPT-3 and PaLM are typically autoregressive, predicting each token based on preceding context, and are trained using massive corpora to learn general language patterns. LLM capabilities are further extended through mechanisms such as in-context learning, reinforcement learning from human feedback (RLHF), and retrieval-based augmentation. The latter, used in this thesis, enhances response quality by injecting external, domain-specific context at inference time, without altering the model’s weights.

**Four Waves of Language Modeling** According to Minaee et al. [19], the development of LLMs can be traced through four distinct phases: (1) statistical models, (2) neural

## 2. BACKGROUND

---

models, (3) pre-trained models, and (4) LLMs.

1. **Statistical models:** Early statistical models, such as n-gram Markov models, estimated word sequences using co-occurrence counts and relied heavily on smoothing to counter data sparsity [20].
2. **Neural models:** Neural models replaced discrete counts with dense embeddings learned from context [21], allowing for semantic generalization across tasks such as search and translation [22], [23].
3. **Pre-trained models:** Pre-trained models like BERT decoupled task-specific supervision from language modeling, using large-scale unlabeled data for transfer to downstream tasks [24].
4. **LLMs:** LLMs extend this pipeline, scaling both parameters and corpora to reach emergent performance regimes [25], [26].

**Transformer Architectures and Scaling** Transformers are neural architectures built around self-attention, a mechanism that models relationships between all tokens in a sequence at once [27]. Unlike recurrent models, which process input step-by-step, transformers handle entire sequences in parallel, enabling more efficient training on large datasets. Most LLMs use a decoder-only variant tailored for autoregressive text generation. Refinements such as rotary embeddings, RMS normalization, and SwiGLU activations have been introduced to improve training stability and speed during inference [28]. These architectural choices affect how well a model performs under fine-tuning, quantization, or hardware constraints.

**Emergent Capabilities** As LLMs grow in size, they begin to exhibit behaviors that weren't explicitly programmed. One example is in-context learning, shown by GPT-3, where the model can solve new tasks by following examples given at inference time—without any additional training [29]. Instruction tuning, used in models like InstructGPT and Flan-PaLM, helps align outputs with user intent by training on curated prompts [30], [31]. Larger models, such as GPT-4 and PaLM-540B, also show stronger reasoning abilities when guided by chain-of-thought prompting [32], [33]. These emergent capabilities make it possible to steer models through prompting rather than retraining—a practical advantage in domains where resources or data access are limited.

## 2.1 Introduction to Large Language Models

---

**Training Data and Domain Coverage** LLMs are typically trained on large-scale, filtered datasets drawn from the web. GPT-3, for example, was trained on 300 billion tokens from sources like Common Crawl, Books, and Wikipedia [29], while PaLM-2 expanded this to 3.6 trillion tokens covering broader domains such as code and dialogue [26], [34]. The LLaMA models, in contrast, rely solely on publicly available data, curated to balance scale with licensing transparency [28]. Although these datasets enable broad generalization, they offer limited coverage of structured institutional content—such as legal texts, education policies, or internal administrative workflows. In such contexts, where accuracy, terminology, and formal structure matter, domain adaptation becomes necessary.

### 2.1.1 LLM Families

LLMs are often categorized into families, each defined by shared architecture, training scale, and development lineage. These families influence how models can be adapted, deployed, or extended in practice. Open families—with transparent design and released weights—are more suitable for local use, especially in privacy-sensitive settings. Closed models, though strong on benchmarks, are typically restricted by opaque training methods and proprietary licensing, limiting their use in controlled environments.

Following the taxonomy proposed by Minaee et al. [19], this section outlines three core families that shape much of today’s model ecosystem: GPT, LLaMA, and PaLM. Each serves as the foundation for a range of derivatives now used across research and applied domains.

**GPT Family** OpenAI’s GPT models are decoder-only, autoregressive architectures trained for next-token prediction. GPT-1 and GPT-2 were released with open weights and code, making them important early resources for research and experimentation. Later versions—GPT-3 (175B), Codex, WebGPT, InstructGPT, ChatGPT, and GPT-4—are proprietary and accessible only via API. GPT-3 marked a shift by demonstrating in-context learning, enabling models to generalize across tasks using prompt examples without retraining [29]. Subsequent models extended this capability: Codex focused on code generation, WebGPT on retrieval-augmented answers [35], [36]. Instruction-tuned variants like InstructGPT and ChatGPT introduced alignment through reinforcement learning from human feedback (RLHF) [30]. However, the lack of open access to models beyond GPT-2 limits their reproducibility, adaptability, and suitability for on-premise deployment.

## 2. BACKGROUND

---

**LLaMA Family** Meta’s LLaMA models are open-weight and intended for broad accessibility. LLaMA-1 and LLaMA-2 span 7B to 70B parameters and are trained on filtered public datasets [28]. They follow a decoder-only Transformer design like GPT, but incorporate refinements such as rotary embeddings, SwiGLU activations, and RMS normalization to improve efficiency on smaller hardware. Ollama’s registry currently supports LLaMA-2 (7B/13B/70B) and LLaMA-3 (1B/3B/8B/70B/405B), including instruction-tuned and multimodal variants, making them practical for local deployment [37].

The release of weights - albeit under non-commercial terms - has led to a growing ecosystem of instruction-tuned derivatives, including Alpaca [38], Vicuna, Guanaco [39], and Koala [40]. These variants adapt the LLaMA architecture for dialogue tasks using relatively modest resources. Architectural transparency, community support, and integration with local tooling were key reasons for selecting LLaMA-3.2 in this system. It provides a stable base for domain-specific, on-premise adaptation.

**PaLM Family** Google’s PaLM models are designed for scale and multilingual coverage. The original PaLM (540B), along with PaLM-2 and Flan-PaLM, are decoder-only Transformers trained on large, high-quality datasets using the Pathways infrastructure [26], [34]. While they perform well on reasoning and generation tasks, the models are closed-source. Weights, training data, and fine-tuning methods are not publicly available, making them unsuitable for on-premise deployment or reproducible research.

**Independent Open Models** Outside the major families, several independent models expand the open-source ecosystem with distinct design goals. Mistral-7B [41], Falcon-180B, and BLOOM emphasize efficiency, scale, and multilingual coverage, respectively [19]. Mistral follows a LLaMA-like architecture but is trained from scratch and released under Apache 2.0. Falcon, trained on RefinedWeb, is optimized for large-scale performance and released under a permissive license. BLOOM, based on the ROOTS corpus, focuses on transparency and language diversity. These projects show that open-weight innovation is no longer confined to a few core lineages; independent releases now play a central role in shaping the model landscape.

**Representative Releases from Major LLM Families** Table 2.1 summarizes selected models from the GPT, LLaMA, and PaLM families. It includes key attributes relevant to deployment: parameter count, release year, weight availability, and licensing terms. Rather

## 2.1 Introduction to Large Language Models

than listing all variants, the table focuses on prominent examples that reflect the diversity of scale and release practices across these three major lineages.

Model Name	Family	Params	Release Year	Open Weights	License
GPT-3 [19]	GPT	125M, 350M, 760M, 1.3B, 2.7B, 6.7B, 13B, 175B	2020	No	Proprietary (OpenAI)
GPT-4 [19]	GPT	1.76T	2023	No	Proprietary (OpenAI)
LLaMA-2 [37]	LLaMA	7B, 13B, 34B, 70B	2023	Yes, non-commercial	Meta Non-Commercial License
LLaMA-3 [37]	LLaMA	1B, 3B, 8B, 70B, 405B	2024	Yes, non-commercial	Meta Non-Commercial License
LLaMA-4 [37]	LLaMA	16×17B (Scout), 128×17B (Maverick)	2025	Yes, non-commercial	Meta Non-Commercial License
PaLM [19]	PaLM	8B, 62B, 540B	2022	No	Proprietary (Google)
PaLM-2 [19]	PaLM	340B	2023	No	Proprietary (Google)

**Table 2.1:** Characteristics of Major LLM Families

The release status and licensing terms across LLM families introduce a more fundamental divide: the open-source versus closed-source paradigm. This distinction is legal and architectural, but it also affects how models are deployed, adapted, trusted, and maintained. Table 2.2 summarizes the operational and strategic trade-offs between these two classes.

## 2. BACKGROUND

---

### 2.1.2 The Open vs. Closed-Source LLM Paradigm

The distinction between open- and closed-source language models shapes how systems are accessed, adapted, and governed. It directly impacts deployment strategy, especially in contexts that require transparency, control, or compliance with internal constraints. While closed models often outperform in benchmarks, open alternatives offer clearer trade-offs in institutional settings where auditability, customization, and privacy are critical.

**Open-Source LLMs** Models like LLaMA, Mistral, Falcon, and BLOOM release their weights, training configurations, and in some cases, data provenance. This enables full customization, including fine-tuning, quantization, retrieval integration, and offline inference. Open models can be deployed on local infrastructure without relying on external services, making them suitable for regulated domains where traceability and control are required. However, they assume local resources and ML expertise. For teams without that capacity, deployment may be complex. In addition, performance on complex reasoning tasks may lag behind state-of-the-art proprietary systems.

**Closed-Source LLMs** Closed models are accessed solely via proprietary APIs. Their internal architecture, training data, and alignment procedures remain opaque. While they deliver strong default performance and are easy to integrate, they offer no access to model internals, no local inference, and no ability to adapt behavior beyond prompting. They are attractive for quick prototyping or where infrastructure is lacking. But they introduce recurring token-based costs, vendor lock-in, and structural limitations on adaptation and deployment.

**The dilemma** Though not deeply theorized in formal literature, this divide is widely debated in practitioner communities. Most academic surveys emphasize model performance; few address governance, legal alignment, or control. Practitioner writing, however, highlights these operational trade-offs as central to real-world decision-making [42]–[44]. To structure this comparison, Table 2.2 summarizes key dimensions relevant to institutional use: transparency, deployment, cost, control, and adaptability.

The comparison shows a consistent pattern: open-source models offer more control, transparency, and deployment flexibility, especially when institutional language, internal data, or compliance requirements are involved [45]. Proprietary models, by contrast, limit adaptation and expose organizations to external dependencies. These constraints are especially relevant in domain-specific settings, where alignment with internal semantics and

## 2.1 Introduction to Large Language Models

governance regimes is critical. The next section explores how domain adaptation techniques address these gaps and enable specialization without compromising deployment boundaries.

Dimension	Open-Source LLMs	Closed-Source LLMs
Transparency	Full or partial access to architecture, weights, training corpus, and logs	No access to architecture, weights, or training corpus
Prompting	Fully supported; same or greater flexibility than closed models	Fully supported; limited by API capabilities
Fine-Tuning	Fully supported	Not possible
Quantization & Optimization	Full control	Not accessible
Inference-Time Control	Fully customizable	Limited: some tool plugins, no custom stack
RAG	Full pipeline customization	Only through vendor UI (e.g. uploading docs to ChatGPT); not controllable
Tool Use / Agents	Full integration	Limited to plugin frameworks provided by vendor
Deployment	Any environment: on-prem, cloud, edge, air-gapped systems	Remote only; runs on vendor infrastructure
Cost Predictability	One-time compute + hardware investment; no per-query costs	Variable or per-token pricing; vendor lock-in
Scalability	Scalable across consumer GPUs, HPC, clusters, edge devices	Not user-controllable
Privacy & Data Sovereignty	Full data locality	User data passes through vendor servers; regulatory misalignment risk
Security / Auditability	Full model access allows auditing, red-teaming, safety tuning	Model internals not inspectable or auditable
Legal & Licensing	Varies by model	Proprietary terms; no redistribution or reverse-engineering
Reproducibility	Supports reproducible research and independent evaluation	No way to replicate training setup or alignment methods

## 2. BACKGROUND

---

Dimension	Open-Source LLMs	Closed-Source LLMs
Adaptability to Domain	Full control over weights, data, tools → easier to specialize	Limited to few-shot prompting; no parameter-level adaptation
Community Ecosystem	Hugging Face, Ollama, LM Studio, vLLM, LangChain, open benchmarks	Closed ecosystems

**Table 2.2:** Comparison of open-source and closed-source LLMs

The comparison highlights a consistent pattern: closed models often lead in benchmarks, but open models offer greater transparency, control, and deployment flexibility. For institutions with domain-specific data or regulatory constraints, this control is critical. Open models allow fine-grained adaptation—through tuning or retrieval—aligned with internal formats, semantics, and governance. The next section explores how such domain adaptation techniques compensate for the limitations of general-purpose pretraining in localized deployments.

### 2.2 Methods for Domain Adaptation

Pretrained LLMs, while broadly capable, often underperform in domain-specific settings where institutional language, structured formats, or procedural knowledge are required. This gap stems from the training data: models like GPT-4o are trained on large, heterogeneous web corpora rather than curated, domain-specific texts [19]. As a result, when asked detailed questions about internal policies, niche entities, or localized procedures, their responses may be vague, speculative, or incorrect. General-purpose training lacks the semantic grounding and contextual familiarity needed for reliable inference in specialized domains.

Domain adaptation addresses this limitation by exposing models to targeted data through methods such as fine-tuning, instruction tuning, or retrieval augmentation. These techniques allow the model to internalize relevant language patterns, constraints, and decision logic, reducing reliance on prompt design alone. In privacy-sensitive environments, adaptation also enables the use of internal data that cannot be shared externally due to legal or ethical restrictions. This thesis explores how such techniques can support relevance, control, and local execution in institutional contexts.

Adaptation is necessary because pretrained LLMs are stateless, probabilistic, and prone to hallucination when operating outside their training distribution. They lack access to



current or private information and are often too large for efficient use in constrained environments. Adaptation narrows the model’s scope and grounds its behavior in verifiable context, improving performance and control in specialized applications.

### 2.2.1 General Overview

In sum, domain adaptation methods can be grouped by the degree of model modification and infrastructure complexity they require. The three most prominent classes include prompt-based methods, retrieval-augmented generation (RAG), and model fine-tuning [19]:

**Prompt engineering** Prompt-based methods guide model behavior through carefully constructed input instructions. They require no modification of model weights and are relatively easy to implement, making them a common starting point for domain adaptation. Prompts are written text that frame the task in ways the model can interpret, often leveraging knowledge of the model’s capabilities and limitations. Prompt design is inherently iterative and exploratory, and can be used alone or alongside other adaptation techniques.

Several structured strategies have been developed to improve prompt effectiveness. One of the most widely used is *Chain-of-Thought (CoT)* prompting, which encourages the model to reason through intermediate steps before arriving at a final answer. *CoT* exists in two main forms: *zero-shot*, which uses general cues like "let’s think step by step", and *example-based*, which embeds explicit reasoning paths in the prompt. While the latter tends to yield better results, it is more labor-intensive to construct. *Tree-of-Thought (ToT)* extends this approach by prompting the model to generate and compare multiple reasoning paths in parallel, selecting the most coherent outcome—a useful mechanism for tasks involving ambiguity or multiple plausible answers.

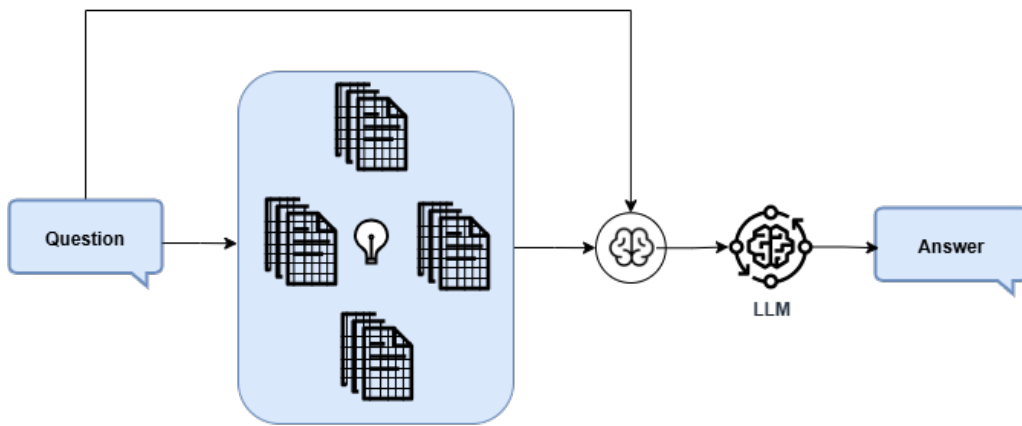
Other approaches target different aspects of control. *Self-Consistency* reduces output variability by sampling multiple completions and aggregating their results. *Reflection* prompts the model to critique and revise its own outputs. *Expert Prompting* positions the model as speaking from the perspective of a specialist, simulating domain-specific knowledge. *Chains* break complex tasks into sequential prompts, while *Rails* constrain output format and content using templates or predefined rules. Finally, *Automatic Prompt Engineering (APE)* uses LLMs themselves to generate, evaluate, and refine prompts automatically based on performance.

## 2. BACKGROUND

---

**Retrieval-Augmented Generation (RAG)** *RAG* combines prompting with access to an external knowledge base, and works by the LLM receiving input retrieved from a corpus (typically vector-indexed internal documents) that is semantically aligned with the user query. This retrieved context is added to the prompt, grounding the response in the given domain data. In sum, the three main components of the system lies in its name: retrieval, augmentation, and generation [46].

- **Retrieval phase:** The system first converts the user query into an embedding and compares it against a pre-indexed collection of vectorized documents. These documents typically consist of semantically segmented internal texts relevant to the domain, such as regulations, policies, or organizational records. The top- $k$  most similar segments are selected based on similarity metrics.
- **Augmentation phase:** Retrieved segments are merged with the user query to construct an augmented prompt. This prompt acts as the model’s context window, allowing it to condition its output on external, task-relevant data without modifying internal parameters.
- **Generation phase:** The augmented prompt is then passed to a language model, which generates a response based on both the retrieved information and the query. The model remains fixed, and the adaptation occurs dynamically at inference time.



**Figure 2.1:** Retrieval and generation flow. Adapted from LangChain documentation [47].

Figure 2.1 illustrates this retrieval-augmented inference loop in abstract terms. A user question is first processed through a retrieval component, which identifies and selects relevant documents from a knowledge base. These documents, represented as semantically

## 2.2 Methods for Domain Adaptation

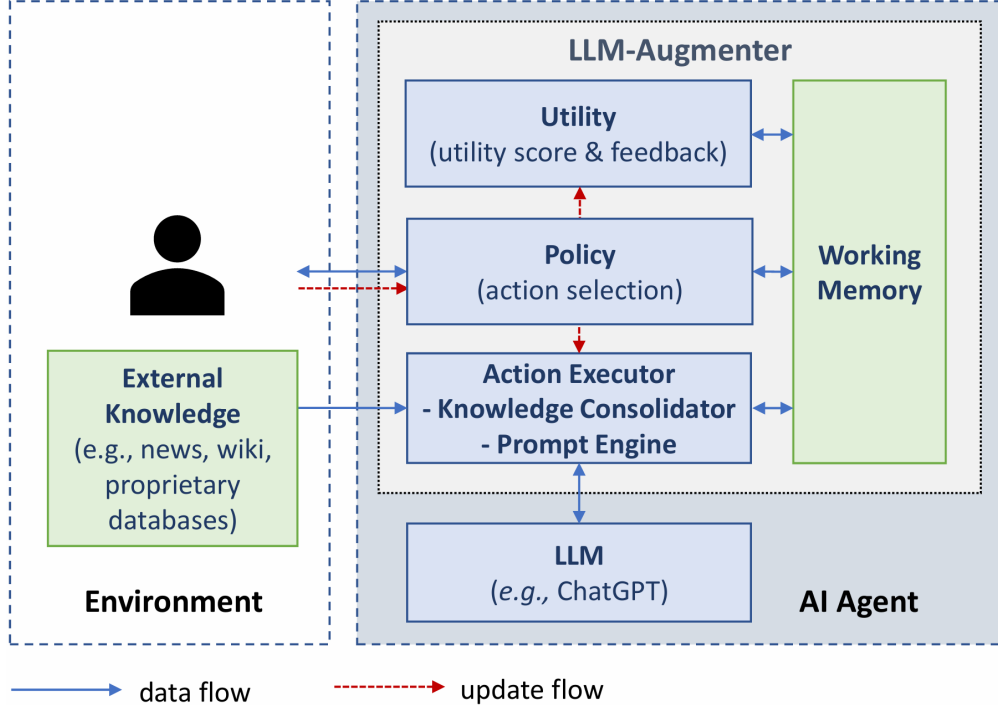
---

indexed chunks, serve as context to augment the original query. The combined input is then passed to the language model, which generates a response conditioned on both the retrieved content and the user’s prompt. This architecture captures the core structure of RAG systems: the model itself remains unchanged, while external knowledge is injected at runtime to guide and constrain its output.

**Tool-Augmented LLMs and Agent Architectures** While retrieval provides one mechanism for augmenting LLMs with external information, it is part of a broader class of strategies in which models interface with external tools. In this context, a “tool” refers to any external function or system, such as APIs, calculators, databases, or file systems, that the model can call to enhance its capabilities. *Retrieval-Augmented Generation (RAG)* is a specialized instance of tool use, where the tool is a semantic search mechanism over a text corpus. More general tool-use enables LLMs to extend beyond static generation, allowing for dynamic data access, computation, or environment interaction [19].

A complementary class of strategies involves modifying the model itself through fine-tuning. Full fine-tuning adjusts all model parameters using domain-specific labeled data, but is computationally expensive and typically requires high-end hardware. More common are parameter-efficient fine-tuning methods, such as *LoRA (Low-Rank Adaptation)*, *prefix tuning*, and *adapter layers*, which introduce a small number of trainable weights while keeping the base model frozen. These methods reduce memory and compute requirements, making them feasible for constrained environments. Fine-tuning enables the model to internalize domain knowledge and task-specific behavior, especially in cases where static prompting or tool use alone is insufficient.

## 2. BACKGROUND



**Figure 2.2:** LLM Agent Architecture. Courtesy of [48].

Figure 2.2 depicts a modular agent framework that integrates external tool use into an LLM-driven decision loop. The architecture consists of four main components: a memory module to track dialog state or intermediate context; a policy module that determines which action to take given the current state; an executor that performs the selected action, such as retrieving external knowledge or calling an API; and a utility module that evaluates outputs based on alignment with task goals or user expectations. This structure enables the model to iteratively reason, retrieve, act, and revise its behavior across multiple steps. While such agent systems can generalize across tasks, their implementation in closed environments is limited by interface complexity, auditability requirements, and control over tool behavior.

While agent-based systems offer general-purpose task coordination through tool integration, their deployment in on-premise settings is constrained by interface complexity, control requirements, and infrastructure overhead. Partial implementations may adopt fixed tool sets, predefined action flows, or minimal reasoning modules. In this context, retrieval-based methods remain a more tractable form of augmentation: they require less orchestration, preserve model determinism, and are easier to audit and constrain. Agent

architectures, while more expressive, are less suited to constrained environments without significant simplification.

### Summary

This chapter introduced the key concepts that inform the rest of the thesis. It outlined how large language models are built, how they differ across families, and how the choice between open- and closed-source models shapes deployment strategy. It also reviewed the main approaches for adapting general-purpose models to domain-specific tasks. Among these, prompt engineering and RAG stand out for their low resource demands and suitability for local use. These methods form the basis for the system developed in this thesis, which is introduced in the next chapter.

## 2. BACKGROUND

---

## 3

# Overview

This chapter introduces the proposed system in broad terms, focusing on its purpose, architecture, components, and user-facing flow. It sets up the technical depth of Chapter 4, Design, by providing a high-level understanding of what is being built and why.

### 3.1 System Overview

The system is a locally deployed assistant for answering domain-specific questions related to university regulations. It integrates document retrieval with controlled prompting to ensure that responses remain grounded in officially ingested source material. All inference and data processing are performed entirely on-premise, without dependence on external APIs or cloud infrastructure.

The system ingests structured policy documents, like enrolment and application regulations, extracts semantic representations of their content, and indexes them in a vector store for similarity search. At runtime, user questions are embedded into the same vector space, and the most relevant document chunks are retrieved and passed to the LLM for answer generation. The language model is explicitly constrained to base its responses solely on these retrieved contexts.

The application stack consists of a lightweight Streamlit frontend, a LangChain-based orchestration layer, and two local Ollama models: a small LLaMA 3-based generative model (`llama3.2:3b`) and an embedding model (`mxbai-embed-large`) for semantic search. ChromaDB is used as the persistent vector store. The architecture is modularized to allow additional documents to be incorporated without modifying the system logic. Prompts, chunking strategies, and model parameters are isolated and configurable, making systematic evaluation and adaptation possible. The system is designed to function within con-

### 3. OVERVIEW

---

strained compute environments while maintaining acceptable responsiveness and accuracy for user queries grounded in a fixed document corpus. A 3-billion-parameter model was selected to accommodate limited local hardware. For improved reasoning, coherence, and response quality, deployment on more capable systems using larger models is recommended.

## 3.2 Use Case and User Interaction Flow

While the architecture supports substitution of both model and data components, the current deployment is centered on policy documents and FAQs from Vrije Universiteit Amsterdam. The use case was motivated by the recurring delays students face when interacting with the university’s student desk, where responses often reference publicly available documents. By automating access to this information, the system aims to reduce workload for staff, shorten student wait times, and demonstrate the feasibility of local LLM infrastructure in institutional settings.

The primary users are current or prospective students seeking procedural information regarding application deadlines, tuition fees, admission requirements, and related administrative topics. These questions are typically addressed by university staff via email, often after several days. The chatbot replicates this functionality using RAG, backed by a local vector store and a constrained language model that is prompted only with grounded document content.

Figure 3.1 illustrates the core interaction flow. Users access the chatbot through a web-based interface served by Streamlit. Upon landing, they are presented with a general prompt input field, a selectable topic dropdown for narrowing query scope (e.g., “Application and Enrolment”), and a short list of example questions to guide usage. The interface clearly indicates that an AI language model is used, notes its limitations, and provides a fallback button to contact the university directly. If the fallback is used, the system can optionally generate a summary of the conversation to help students formulate a more precise inquiry.

All user input is processed as free-text queries. When a query is submitted, it is embedded using a local embedding model and compared against document chunks stored in ChromaDB. The top matches are ranked by similarity score, and if the best match exceeds a predefined threshold, the associated context is passed to the generative model alongside the user’s original question and previous dialogue turns. The output includes a natural language answer followed by an expandable citation panel listing the document title, page number, reference snippet, and link. If no match meets the threshold, the model



## 3.2 Use Case and User Interaction Flow

---

is prompted to return an explicit fallback message indicating that it could not confidently answer the query.

Session memory is maintained throughout the chat to support follow-up questions. The model receives a trimmed conversation history to preserve context while respecting token limits. In all cases, responses are strictly grounded in retrieved content, with prompt instructions designed to suppress hallucination and reject unsupported answers.

### 3.2.1 Student Flow

This subsection focuses students and prospective students and traces the complete user journey from system entry to resolution or escalation. The corresponding interaction flow is shown in Figure 3.1.

### 3. OVERVIEW

---

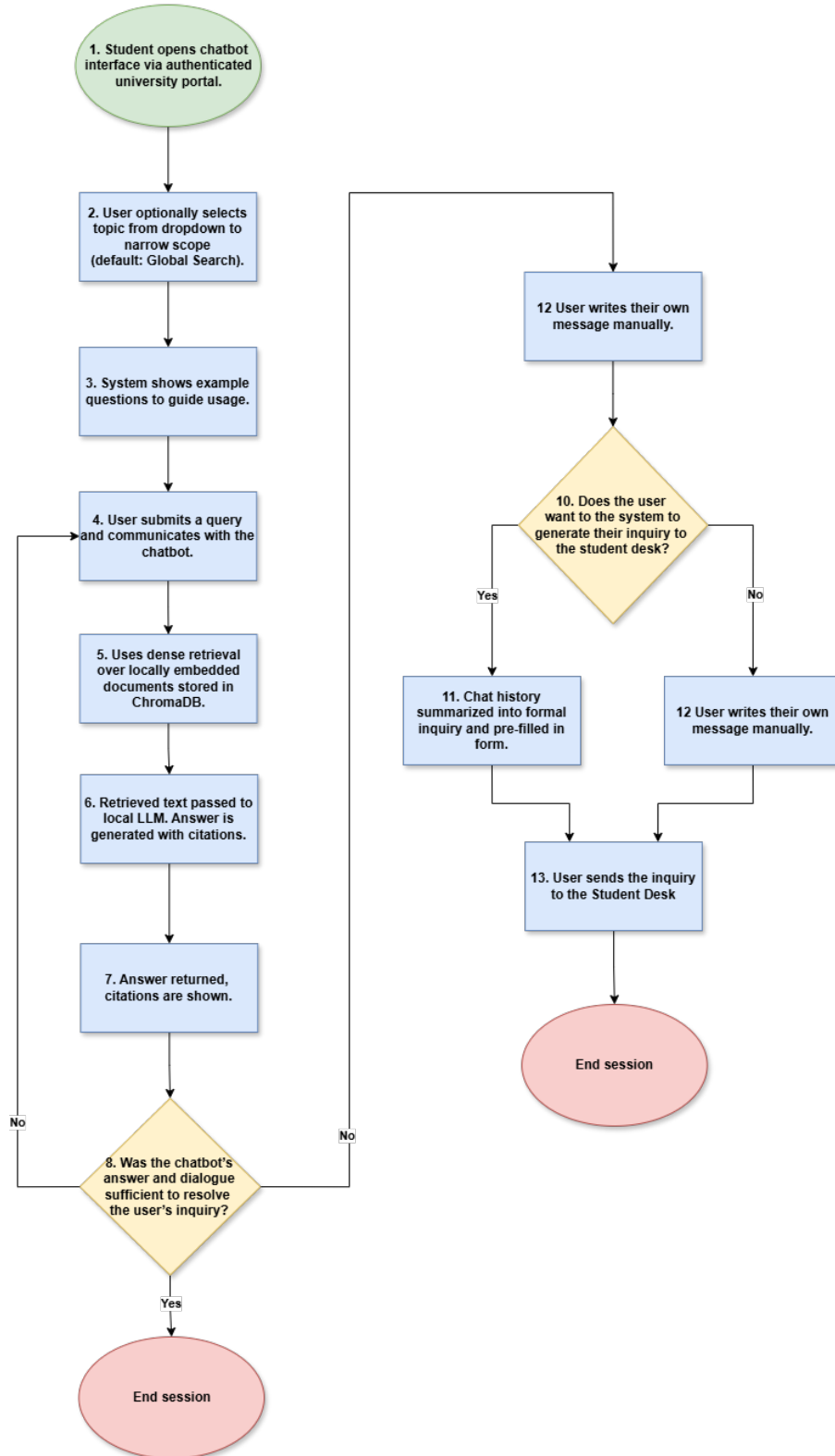


Figure 3.1: (Prospective) Student User Flow

## 4

# Design

This chapter explains the rationale behind the system’s design choices, covering structural decisions, modularity, and deployment strategy. It connects design goals to implementation details, preparing for the next phase of technical analysis.

### 4.1 Design Rationale and Goals

The system is designed to demonstrate how open-source LLMs can be adopted and adapted for fully local deployment. Its architecture, component choices, and execution model are all shaped by the central goal of addressing **RQ1.1**—namely, how organizations can achieve data sovereignty, privacy compliance, and domain-specific performance without reliance on proprietary LLM services.

The selected use case is illustrative rather than limiting. The architecture is domain-agnostic, with modular components and swappable interfaces designed to generalize across institutional deployments. The guiding principles are locality, modularity, transparency, and constrained reasoning, each selected to support one or more of the thesis sub-questions.

#### Research Questions as Design Drivers

- **RQ1.1** guided the selection of all system components: a 3B parameter open-weight model served via Ollama, a dense embedding model for vector search, and a lightweight vector store (Chroma) for on-premise indexing. All components are open source and locally deployed, ensuring architectural transparency and allowing experimentation beyond the limits of proprietary APIs.
- **RQ1.2** shaped the ingestion and retrieval pipeline. Policy documents are segmented and embedded into topic-specific indexes, which support domain-relevant retrieval.

## 4. DESIGN

---

Prompt formatting and chunking are tuned to maintain source grounding and reduce context leakage. This setup supports document alignment and secure integration of internal content into the retrieval-augmented generation flow.

- **RQ1.3** informed trade-offs between model scale, latency, and deployment constraints. The use of a quantized 3B model balances performance and efficiency, allowing inference without a GPU. The system architecture mirrors core functionality offered by commercial LLM APIs, while demonstrating under what conditions open models offer a viable and privacy-preserving alternative.

The mapping between each design goal and its corresponding implementation strategy is summarized in Table 4.1. This table defines the core priorities embedded into the system architecture and how each is operationalized at runtime.

Design Goal	Mechanism in System
Data sovereignty	No remote APIs; all inference and storage are local
Domain adaptation	Per-topic document indices; isolated retrieval pipelines
Low-latency inference	Quantized 3B model served via Ollama; fast local embedding
Transparency	Prompt-grounded generation with inline citations and traceable chunks
Modularity	Decoupled embedding, retrieval, prompting, and UI components
Deployment flexibility	All dependencies are local; no internet connection required

**Table 4.1:** Design goals and their implementation in the system architecture.

### 4.2 System Architecture and Data Flow

Figure 4.1 provides a high-level overview of the system architecture and data flow. It separates the document ingestion pipeline, used to process and embed regulatory documents prior to runtime, from the main execution-time components that handle user interactions

## 4.2 System Architecture and Data Flow

and query resolution. Each numbered step in the diagram corresponds to a distinct data or control flow within the system. The following list describes the key phases of the pipeline.

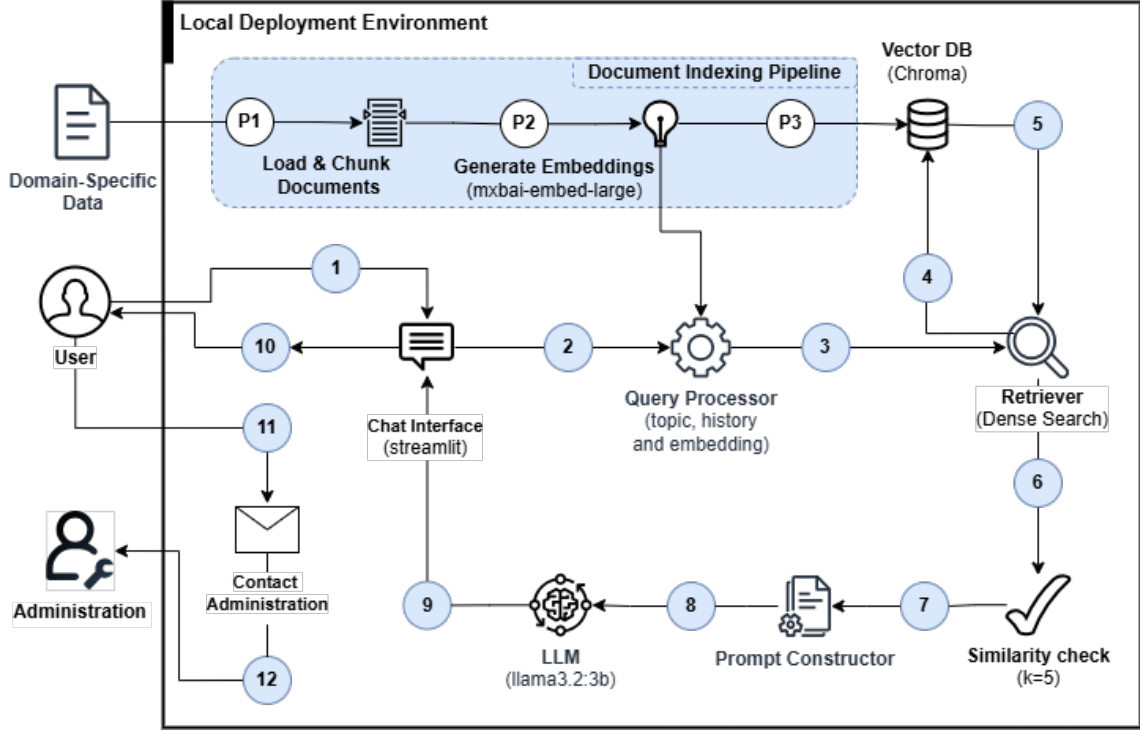


Figure 4.1: High-level System Architecture

### Document Ingestion Pipeline (Steps P1–P3)

Before deployment, all relevant regulatory PDFs are processed through an offline ingestion pipeline. This stage is executed once per update to the document corpus:

- **P1 – Load and Chunk Documents:** PDF files placed in the `data/` directory are parsed and segmented into overlapping text chunks using a fixed-size window. Overlap ensures contextual continuity between adjacent chunks.
- **P2 – Generate Embeddings:** Each chunk is transformed into a dense vector using the local embedding model `mxbai-embed-large`, served via Ollama. All computation occurs locally, with no external API calls.
- **P3 – Index into Vector Store:** The embeddings are stored in a local ChromaDB instance under topic-specific collections. Chunk-level metadata (e.g., filename, page number, position) is preserved for use during response citation.

## 4. DESIGN

---

Once indexed, the system can answer queries without reprocessing the documents unless the underlying corpus is modified.

### Runtime Interaction Flow (Steps 1–12)

1. **User Input:** The user enters a natural language query into the web-based Streamlit chat interface. A dropdown menu allows optional scoping to a document category.
2. **Query Processor:** The query, topic scope, and trimmed chat history are received by the backend. The query is embedded using a local embedding model (`mxbai-embed-large`), and relevant metadata is attached.
3. **Embedding Dispatch:** The computed query vector is passed forward for similarity-based retrieval.
4. **Retriever (Dense Search):** The retriever performs a dense similarity search over the ChromaDB vector store.
5. **Top-k Selection** The top 5 document chunks are retrieved based on cosine similarity and passed to the next stage for filtering and prompt construction.
6. **Similarity Check:** A hard threshold is applied to the similarity score of the top-ranked chunk. If it falls below this threshold, no context is provided to the model, and a fallback message is generated indicating that the system cannot confidently answer the query.
7. **Prompt Constructor:** If context is valid, the top chunks are concatenated with the user query and prior chat history to construct a structured prompt. Grounding instructions are included to limit generation to retrieved evidence only.
8. **LLM (llama3.2:3b):** The prompt is passed to a locally running open-source language model via Ollama. The model returns a natural language answer, constrained by the contextual evidence. The generated response is returned to the frontend for display to the user.
9. **Response Return:** The generated response is returned to the frontend for display within the chat interface.
10. **View message:** The generated response is being read by the user, and can now decide to continue conversation or contact the student desk if needed.

11. **Contact Administration:** The user is offered the option to forward their question to the university administration via a built-in contact form. The system can automatically generate a well-formed inquiry message based on the user’s original question and chat history, which the user can review and submit.
12. **Send Inquiry:** The user chooses to either submit the automatically generated message (based on the prior conversation) or manually write their own. The finalized inquiry is then forwarded to the administration through the contact interface.

While the chat interface is accessed through a web browser, no data is transmitted beyond the host machine. New domain-specific documents can be incorporated at any time by adding them to the `data/` directory and rerunning the ingestion pipeline—no code modifications are required.

## 4.3 Component-Level Design Decisions

This section details the concrete design decisions made across key system components. While the implementation is original, elements of the architecture were initially informed by community tutorials on local RAG systems, particularly those by Dutch YouTuber Thomas Janssen [49], [50]

Two tutorials in particular were used as early reference points: one focused on end-to-end RAG with Ollama and LangChain using web-scraped content [51], [52], and another demonstrating a PDF-based RAG chatbot with OpenAI APIs [52], [53]. While the current system diverges significantly, particularly in modularization, how RAG is organized, and evaluation-design, the ingestion script remains structurally similar to the original example.

### Model Selection and Quantization

The generative model used in the deployed system is `llama3.2:3b`, a 3-billion parameter instruction-tuned model served locally through Ollama. This model was selected for its balance between generative capability and hardware feasibility. It supports quantized inference with acceptable latency on CPU-only machines, making it well suited for edge or constrained environments.

Only models below 7B parameters were considered, constrained further to 4-bit and 5-bit quantized variants. These thresholds were selected to ensure compatibility with development machines limited to 8GB RAM and to reduce cold-start latency. Larger models were excluded due to extended load times, increased token throughput cost, and

## 4. DESIGN

---

memory instability in longer sessions. Quantization was therefore critical to achieving near-interactive response times without relying on GPU acceleration or distributed inference.

### Embedding Strategy and Vector Indexing

For computing semantic similarity between queries and document chunks, the embedding model `mxbai-embed-large` was used. This transformer-based model was selected based on comparative performance in public benchmarks and informal tests, where it showed robust semantic generalization on longer, policy-like text. Lighter alternatives such as `all-MiniLM` were tested but discarded due to semantic fragmentation in longer documents and lower recall in multi-sentence queries.

Embeddings are generated during offline ingestion to avoid runtime overhead. Chunked documents are embedded using `OllamaEmbeddings` and indexed using ChromaDB, a lightweight vector store compatible with LangChain. Chroma was selected for its simplicity, persistence support, metadata filtering, and zero-server architecture. Alternatives such as Qdrant and Weaviate were ruled out due to their heavier dependencies and server requirements, which were deemed unnecessary in a single-user offline deployment. The ingestion process builds both topic-specific and global indexes, supporting scoped retrieval while preserving fallback coverage.

### Prompt Construction and Grounding

Prompt construction is handled through a static template composed of three segments: system-level instruction, truncated dialogue history, and top-k document chunks retrieved from ChromaDB. The system prompt enforces strict grounding by explicitly directing the language model to disregard any knowledge not contained in the retrieved content. This is reinforced through small chunk sizes (300 characters with 100-character overlap) and early exit logic when retrieval scores fall below a threshold. Dynamic or chain-based prompting was deliberately avoided to reduce complexity and improve determinism during generation.

### User Interface and Memory

The interface is built with Streamlit, chosen for its minimal setup, built-in state management, and seamless integration with Python-based backends. Conversation state is maintained using `st.session_state`, enabling multi-turn interactions without requiring external session storage. Citation formatting in the interface is driven by metadata persisted during ingestion, allowing traceable, document-grounded answers. Fallback behavior



### **4.3 Component-Level Design Decisions**

---

is explicitly surfaced to the user when no confident match is retrieved, supporting transparency and reducing the risk of hallucinated output.

#### 4. DESIGN

---

## 5

# Implementation

This chapter details the technical implementation of the chatbot system developed for this thesis, which is available as open-source code at [54]. It outlines how institutional documents are embedded into a local vector database and queried via a locally hosted LLM to support question answering over private, domain-specific content. The system operates in both graphical (GUI) and headless modes. Documents are preprocessed offline into persistent Chroma vector stores using local embeddings. At inference time, the engine retrieves semantically relevant chunks, assembles a constrained prompt, and invokes the local LLM. The output includes both the generated response and a corresponding source attribution. The following sections describe the system architecture, data processing pipeline, inference logic, user interface, and supporting infrastructure.

## 5.1 Overview and Codebase

The system architecture is organized around four logically separated components: document ingestion, retrieval and generation, user interaction, and evaluation. Each operates independently and communicates through clearly defined interfaces. The evaluation framework runs separately from the interactive system and is used to systematically assess performance across predefined test cases. All modules are implemented in Python, with minimal external dependencies and environment-based configuration for model selection. Figure 5.1 outlines the directory structure.

## 5. IMPLEMENTATION

---

```
.
|- ui/
|   |- chatbot_app.py           # Streamlit front-end interface
|- core/
|   |- rag_engine.py           # RAG pipeline: retrieval, prompting, LLM call
|   |- config.py               # Centralized model and system config
|   |- __init__.py
|- evaluation/
|   |- test_cases/             # JSONL files with test queries
|   |- results/                # Per-system test outputs and summaries
|       |- laptop/             # Result JSONs from test_script (e.g. --dir laptop)
|       |- desktop/            # Result JSONs from other systems
|   |- test_script.py          # Runs evaluation test cases
|   |- llm_judge.py            # LLM-based answer quality evaluator
|   |- evaluate_results.py      # Aggregates and summarizes evaluation results
|   |- aggregate_metrics.csv    # Global summary across systems
|- data/                        # VU source documents (chunked + raw)
|- chroma_db/                  # Auto-generated Chroma vector DB
|- ingest_database.py           # Document parsing and indexing pipeline
|- question_generation.py       # Tooling for draft test question generation
|- email_handler.py             # Stub module for possible email interface
|- .env                         # Local environment config (if used)
|- .gitignore
|- README.md
|- requirements.txt
```

**Figure 5.1:** Codebase layout with functional annotations.

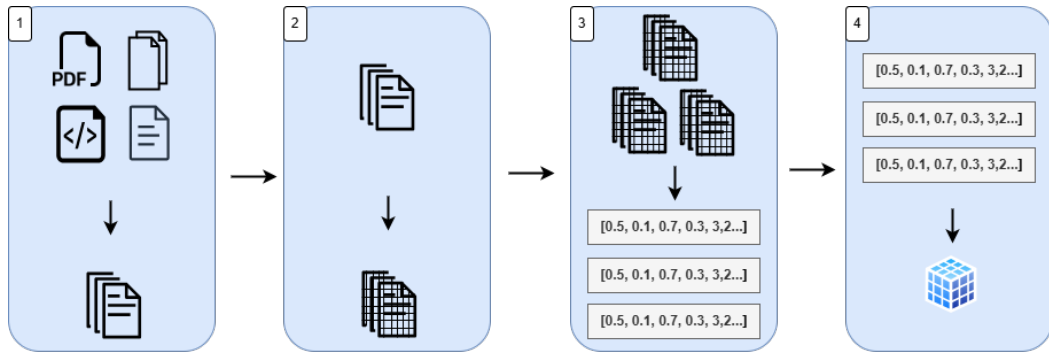
Preprocessing logic resides in a top-level script (`ingest_database.py`), which is executed each time new or updated documents are added to `data/`, in order to regenerate the vector store. The core retrieval and generation logic is isolated in `core/rag_engine.py`, with a single entry point (`run_query()`) that accepts a query, topic label, and optional chat history. This function is reused across both interactive and evaluation contexts.

The Streamlit interface (`ui/chatbot_app.py`) serves as the primary user-facing entry point. It handles topic selection, chat state, and optional support requests, delegating all retrieval and generation to the backend engine. Auxiliary modules such as `question_generation.py` and `email_handler.py` implement optional logic for summarizing conversations and forwarding inquiries to human support channels.

Evaluation logic is placed under the `evaluation/` directory, making structured and reproducible testing of system possible. Predefined test cases are stored as JSONL files, each specifying a topic, input query, and expected behavior. The evaluation runner (`test-script.py`) executes these cases and logs the system’s responses, which are subsequently assessed for correctness via automated heuristics or LLM-based scoring (`llm_judge.py`). Aggregation and summary scripts (`evaluate_results.py`) compute performance metrics across cold and warm start scenarios, facilitating comparative analysis across hardware, settings, and model variants.

## 5.2 Data Ingestion and Preprocessing

Before inference can occur, all relevant university documents must be parsed, segmented, and embedded into a searchable vector space. This process is implemented in `ingest_database.py` and follows four sequential stages, as shown in Figure 5.2.



**Figure 5.2:** Ingestion pipeline with four core stages. Adapted from LangChain documentation [47].

1. **Load:** Raw PDFs are sourced from topic-specific subdirectories under `data/`. Each topic corresponds to a semantic domain (e.g., application enrolment, selection policy) and is processed independently. Documents are parsed using `PyPDFDirectoryLoader` into a standardized internal format.
2. **Split:** Documents are segmented using a recursive character splitter with a chunk size of 600 characters and an overlap of 100. This level of granularity retains local context while keeping chunks within practical prompt limits during retrieval and inference.

## 5. IMPLEMENTATION

---

3. **Embed:** Text chunks are converted into dense vector representations using `mxbai-embed-large`. Embeddings are computed through LangChain’s `OllamaEmbeddings` interface and are generated offline. The embedding model identifier is declared in the `.env` file.
4. **Store:** Vectors are indexed using ChromaDB, a lightweight embedded vector database optimized for local storage and retrieval. Each topic forms its own collection in `chroma_db/`, and a global collection is built by aggregating all chunks across domains. UUIDs are assigned to each chunk to support traceable retrieval during inference.

### 5.3 Retrieval and Generation Engine

The retrieval and generation logic lies within `rag_engine.py`, which is a module that handles semantic search over precomputed embeddings, prompt construction, while also logging system diagnostics and latency for each execution.

#### Retrieval Pipeline

At inference time, users explicitly select the topic associated with their query. This selection determines which vector store is used for retrieval. If a specific domain is chosen (e.g., `selection_placement`), only that domain’s Chroma collection is queried. If the user selects `Global Search`, the system performs retrieval over the full index, which includes all embedded documents.

For each query, the retriever fetches the top- $k$  most relevant chunks ( $k = 5$ ) using semantic similarity scoring. Retrieved chunks are concatenated to form the basis for the LLM prompt. Retrieved documents are deduplicated based on approximate string matching to avoid near-identical content. From the deduplicated results, the top 2 unique chunks are selected and used to construct the LLM prompt. Each chunk’s source metadata (filename, page number) is preserved to support traceability in the response.

The prompt is constructed by combining the user query, the selected context chunks, and optionally the chat history (capped at three previous turns). This prompt is then passed to the LLM, which generates a grounded response. The generation parameters are defined in the system configuration: `LLM_TEMPERATURE=0.2`, `LLM_TOP_P=1.0`, and `LLM_REPEAT_PENALTY=1.05`. Figure 2.1 in Chapter 2 illustrates this retrieval-augmented inference loop.

The `run_query()` function executes the full inference pipeline: it selects the target Chroma collection based on user input, performs similarity search with  $k$ , constructs the prompt using the top 2 unique chunks, and invokes the LLM. At each stage (retrieval,

prompt construction, and generation) it logs latency and metadata for better evaluation and traceability.

### Prompt Construction and LLM Invocation

For each query, the system constructs a prompt using a constrained instruction format designed to enforce grounding in the retrieved documents. The prompt explicitly instructs the model to generate responses based solely on context passages drawn from university regulations, without relying on prior knowledge. Prompt construction is handled by the function `build_prompt()`. It accepts the user query, up to two deduplicated context chunks, and an optional chat history. The resulting prompt has the following structure:

```
You are a student-facing chatbot answering questions strictly using official
documents from Vrije Universiteit Amsterdam (VU).
Do not use prior knowledge.
Answer the question using only the source material provided.
Be direct and natural.
Do not hedge when the answer is stated clearly.
[Question] {query}
[Source] {chunks[0]}

[Additional Context] {chunks[1]}    # Included only if non-empty

[Conversation History] {chat_history}    # Included only if present
```

If a second chunk is a near-duplicate of the first, it is omitted. Chat history is included only if warm mode is active and limited to the last three turns. The prompt is passed to an instance of `OllamaLLM`, a local wrapper over the model specified in the `LLM_MODEL` environment variable (default: `llama3.2:3b`).

Model instantiation is managed by `get_llm()`, which supports both cold (stateless) and warm (stateful) execution. In warm mode, the model instance is reused across turns, preserving conversational context and reducing latency.

### System Monitoring and Traceability

The inference engine emits structured runtime diagnostics via terminal logs to support performance profiling, debugging, and reproducibility. Logging is active by default and includes detailed measurements for each inference cycle.

## 5. IMPLEMENTATION

---

Key metrics and outputs include:

- Model configuration: `temperature`, `top_p`, and `repeat_penalty`
- LLM initialization time in milliseconds (cold mode only)
- Retrieval time (`retrieval_time_ms`)
- Generation time (`generation_time_ms`)
- Total latency (`total_time_ms`)
- Prompt length in characters
- Similarity scores for retrieved chunks (`k=5`), including source metadata
- System information: OS, CPU, RAM, Python version, GPU memory and utilization
- GPU memory usage before and after model invocation (via `nvidia-smi`)

System-level information is logged on first model instantiation using `psutil` and subprocess access to `nvidia-smi`. The GPU status log reports both static memory usage and utilization percentage, enabling verification of hardware acceleration.

### Excerpt of terminal output:

```
[DEBUG] LLM config → temperature: 0.2, top_p: 1.0, repeat_penalty: 1.05
[DEBUG] LLM initialized in 1949.76 ms | ID: b66d2da6-7071-4eba-b24f-d9e36f514f69
[SYSTEM] OS: Windows 10 | CPU: Intel64 | RAM: 15.84 GB (6.33 GB available)
[GPU 0] NVIDIA GeForce MX250 | 0 MB / 2048 MB used | Utilization: 1%
[INFO] GPU detected in use by Ollama or Python
[TRACE] >>> Entered run_query()
[DEBUG] Retrieved 2 unique chunks (after deduplication).
[CHUNK 1] Score: 0.4512
[CHUNK 2] Score: 0.5034
[DEBUG] Prompt length: 1790 characters
[GPU LOG] BEFORE LLM INFERENCE: 639, 0
[GPU LOG] AFTER LLM INFERENCE: 1153, 96
[LATENCY] Retrieval: 4258.60 ms
[LATENCY] Generation: 24990.63 ms
[LATENCY] Total: 31274.17 ms
```



### Modularity and Execution Interface

The central entry point to the inference pipeline is the function `run_query(query, topic, chat_history=None, log=True, llm=None)`, which performs retrieval, prompt construction, and LLM invocation. It returns a structured dictionary containing:

- `response` (model output)
- `retrieved_docs` (LangChain document objects)
- `retrieved_sources` (source metadata, e.g., filename)
- `similarity_scores` (semantic match confidence for top-2)
- `retrieval_time_ms, generation_time_ms, total_time_ms`
- `sources_display` (snippet-formatted citations for UI rendering)
- `prompt, prompt_length` (constructed LLM input and character count)
- `timestamp` (ISO 8601 format)

The engine is directly invoked by both the Streamlit frontend (`chatbot_app.py`) and the evaluation framework (`test_script.py`).

## 5.4 User Interface

The user-facing interface is implemented using Streamlit and encapsulated in `chatbot_app.py`. The interface provides a lightweight frontend that interacts directly with the retrieval and generation engine.

## 5. IMPLEMENTATION

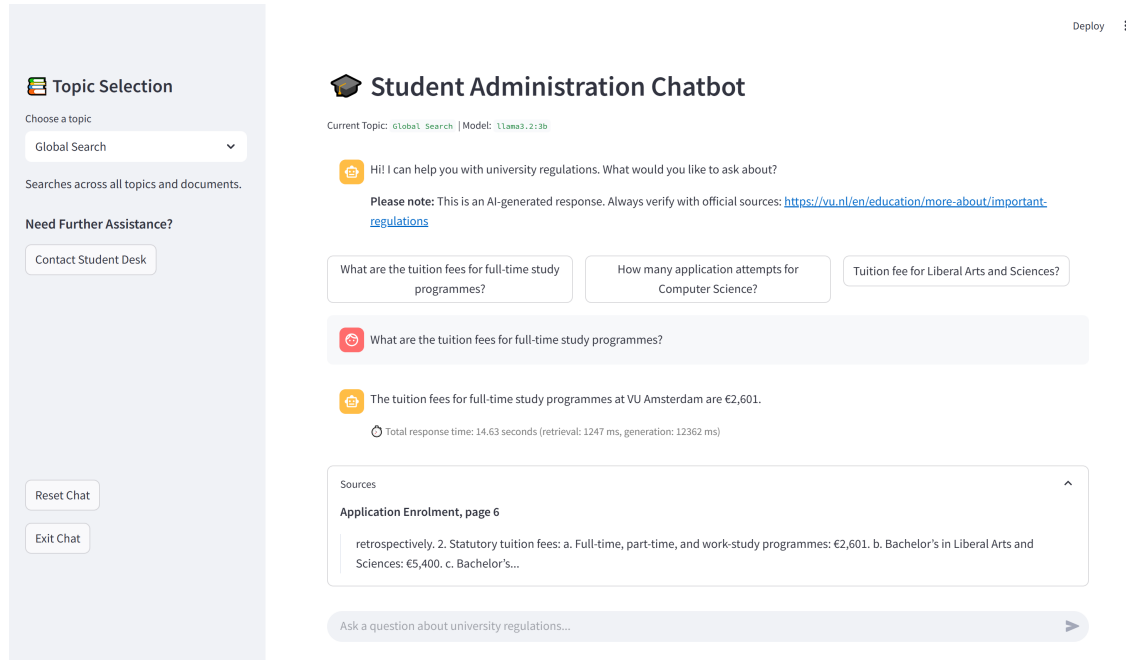


Figure 5.3: Chatbot interface

### Interaction Flow

A topic selector is placed in the top-left corner of the screen (see Figure 5.3). Each topic corresponds to a specific domain and includes a short description to guide the user’s choice. In addition to domain-specific options (e.g., **Application and Enrolment**, **Selection and Placement**), a global fallback mode (**Global Search**) allows for cross-domain queries.

Natural language queries are submitted to the chat input field, and three example queries are available as one-click buttons. System responses are generated in real time and rendered with the corresponding source excerpts shown in an expandable section. **Reset Chat** and **Exit Chat** buttons are also available in the sidebar to clear session state or exit the chat.

Chat history is passed to the RAG engine on every turn to support contextual continuity. To constrain prompt length and avoid latency spikes, only the three most recent user–assistant message pairs are retained. This truncation is enforced within the `run_query()` function when a warm-mode LLM instance is reused. The decision is grounded in empirical evaluation: preserving full chat history across turns resulted in significantly increased inference time and degraded response quality due to prompt overflow. Limiting history depth maintains coherence while controlling prompt size and model load.

### Support Request Functionality

In addition to real-time automated responses, the interface includes a secondary interaction path that enables users to contact human support. This functionality is accessed via the sidebar under “*Need Further Assistance?*” and is designed to assist users who require escalation beyond the system’s current scope.

Upon activation, users are presented with a support form in which they can either manually write a question or opt to generate one automatically based on their prior interaction with the chatbot. The automatic inquiry synthesis is triggered by clicking the **Generate Question** button. Internally, this invokes the function `generate_inquiry_from_history()`, defined in a separate support module. The function receives the full chat history and an LLM instance as input, and returns the inquiry formulated as if written by the user.

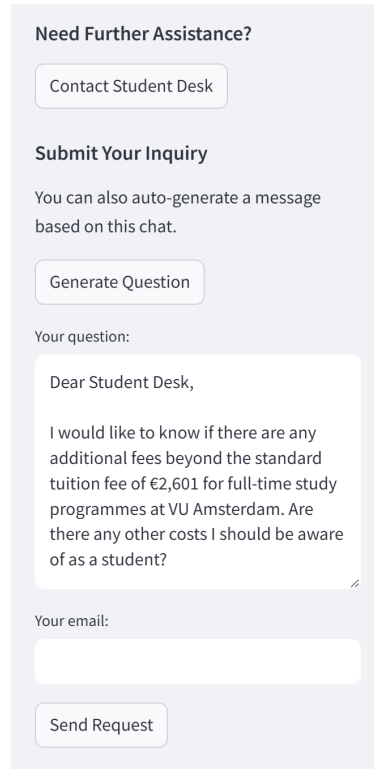
```
"Read the following conversation between a student and an assistant.
Based on the student's questions, write a refined inquiry for the Student Desk
that summarizes what the student is seeking. You may use up to two sentences,
and your message should begin with 'Dear Student Desk,'.
Only include points that the student has actually mentioned or asked about.
Do not invent or add information that was not part of the conversation."
```

The generated inquiry is inserted into a text area where the user can review and optionally edit it. Once finalized, the inquiry and email address are passed to `send_email_to_student_desk()`, a placeholder function for future integration with institutional support services.

Figure 5.4 shows an example of a generated inquiry based on a prior multi-turn interaction.

## 5. IMPLEMENTATION

---



Need Further Assistance?

Contact Student Desk

Submit Your Inquiry

You can also auto-generate a message based on this chat.

Generate Question

Your question:

Dear Student Desk,

I would like to know if there are any additional fees beyond the standard tuition fee of €2,601 for full-time study programmes at VU Amsterdam. Are there any other costs I should be aware of as a student?

Your email:

Send Request

**Figure 5.4:** Automatically generated support inquiry based on chat history.

### System Feedback and Diagnostics

The interface also displays key runtime metadata for each query. Model configuration is displayed below the chat header, and generation time is rendered in milliseconds beneath each response. Retrieved sources are shown in an expandable section following each response. These citations are constructed from metadata attached to the retrieved document chunks and are intended to provide minimal but sufficient provenance. Specifically, the filename and page number are extracted from each chunk’s metadata field (`doc.metadata['source']`, `doc.metadata['page']`), and the first 180 characters of the chunk content are displayed as a preview.

Only the top-ranked document is cited by default, though up to five documents are retrieved internally. This information is included in the returned response dictionary under `sources_display`, alongside similarity scores and full chunk contents. The citation mechanism helps mitigate hallucination by explicitly anchoring generated answers to their underlying sources. Figure 5.3 shows how this feedback is rendered in the user interface .

## 5.5 Evaluation Infrastructure

A custom evaluation framework was implemented to benchmark the RAG pipeline across modes, domains, and hardware configurations. While not fully deterministic due to model variability, the framework is interface-agnostic and supports structured, repeatable testing under controlled conditions.

### Batch Test Execution

Test cases are defined in `.jsonl` format. Each line specifies:

- `question` (natural language query)
- `expected_answer` (reference text for semantic comparison)
- `topic` (used to select the Chroma collection)
- `keywords` (optional lexical constraints)
- `expected_source` (filename or substring)

Execution is handled by `test_script.py`, which invokes the `run_query()` function and logs all results. Tests can be run individually or in batch via folder traversal. Mode selection is controlled via the `--mode` flag:

- `cold` (default): Each query uses a new LLM instance, with no retained history.
- `warm`: A persistent LLM instance is reused across turns; history is truncated to the last three message pairs.

The script allows concurrent specification of test files and execution mode, supporting direct flag-based configuration (e.g., `--mode warm --dir desktop`). Output is saved as JSON in `evaluation/results/<target>/`, with per-test fields logged for analysis.

### Evaluation Metrics

Each result entry includes:

- `total_time_ms`, `retrieval_time_ms`, `generation_time_ms`
- `similarity_scores` for retrieved chunks
- `actual_response`, `retrieved_sources`

## 5. IMPLEMENTATION

---

- Boolean flags for `matched_keywords`, `matched_source`
- Semantic judgment via `llm_judge_passed`

Semantic evaluation is performed by a second LLM instance instantiated through `get_judge_llm()`. The judge returns a binary decision (`true/false`) along with a one-sentence justification. The prompt template ensures consistent judgment criteria across test runs.

### Aggregation and Reporting

Summary generation is handled by `evaluate_results.py`, which parses result files and computes:

- Pass rate (%)
- Mean and standard deviation of total latency
- Mean retrieval and generation time
- Mean similarity score

Results are grouped by topic and mode, and written to `<target>_summary_metrics_timestamp.csv` and `<target>_aggregate_metrics_timestamp.csv`.

## 6

# Evaluation

This chapter presents the experimental design and empirical results used to evaluate the RAG-based chatbot system. The evaluation aims to quantify the assistant’s retrieval accuracy, response correctness, latency behavior, and overall reliability when executed in a fully local environment. Tests were conducted on two machines: (i) a standard laptop equipped with an NVIDIA MX250 GPU and an 8-core Intel CPU, and (ii) a 24-thread gaming desktop with a GTX 1080 Ti GPU. Although the two hardware profiles reflect available rather than representative infrastructure, they illustrate how performance scales with local compute capacity. Despite not mirroring typical university setups, the results demonstrate meaningful latency variation, underscoring the system’s adaptability to different resource conditions. This suggests that more capable hardware could feasibly support larger models locally without sacrificing privacy or responsiveness.

LLMs can be executed on either CPUs or GPUs, but the underlying architectures differ significantly in how they process model inference. CPUs are optimized for sequential general-purpose tasks and offer limited parallelism, which often results in slower inference times for transformer-based models. In contrast, GPUs are designed for massively parallel computations, making them well-suited for matrix-heavy operations such as attention mechanisms and feed-forward layers. Most open-source LLM toolchains, including Ollama and Hugging Face’s ‘transformers’, support GPU acceleration via NVIDIA’s CUDA (Compute Unified Device Architecture), which provides a runtime API and compiler for executing operations on NVIDIA GPUs [55]. Enabling CUDA typically leads to substantial performance gains, especially in generation latency

Initially, evaluations were run on the laptop without CUDA installed, resulting in an average generation latency of approximately 20 seconds. With CUDA support enabled, latency dropped to 15 seconds. On the desktop with proper GPU utilization, generation

## 6. EVALUATION

---

latency further dropped to an average of 0.8 seconds—a 94.7% reduction compared to the CPU-only baseline. This demonstrates how open-source LLMs, even at modest scale, can achieve near real-time responsiveness when deployed on appropriate hardware.

### 6.1 Experimental Setup and Design

The system’s high-level design was originally split across two scripts: `ingest_data.py`, which parses and embeds the regulatory PDFs into ChromaDB, and `chatbot.py`, which bundled both RAG logic and the Streamlit user interface. While simple and functional, this non-modular architecture hindered systematic evaluation. To address this, the UI was fully decoupled and moved to a dedicated frontend script, allowing direct CLI access to the retrieval engine, and separation enabled deterministic testing and automated benchmarking via `test_script.py`.

Evaluation is performed by sending structured test batches, defined in `.jsonl` files, to the RAG engine. Each test file contains ten queries focused on a specific topic domain (`application_enrolment`, `selection_placement`, or `global`). Each query includes a natural-language question, a list of expected keywords, a quoted answer fragment from the original document, and the name of the expected source file. All tests are run in both cold and warm modes. In cold mode, the LLM is re-initialized for every query and chat history is disabled. In warm mode, a single LLM instance is reused with a capped three-turn chat history. Surprisingly, early warm runs exhibited higher latency than cold runs, despite avoiding repeated initialization. This was traced to unchecked chat history accumulation, and resolved by truncating the memory window. Once capped, warm runs consistently outperformed cold runs by an average of about 5 seconds per query.

Test execution is initiated via the command line using `test_script.py`. Input can be a single `.jsonl` file or a directory containing multiple test sets. Execution mode is specified via the `-mode` flag; if omitted, `cold` mode is used by default. Results are saved in the subdirectory defined by `-dir` (e.g., `evaluation/results/desktop/`). Each run produces a structured JSON file containing per-query outputs and metrics. After testing, the `evaluate_results.py` script aggregates all JSON files in the target directory. It produces two outputs: a detailed per-file summary (`<dir>_summary_metrics_timestamp.csv`) and a grouped aggregate overview (`<dir>_aggregate_metrics_timestamp.csv`). Each query is evaluated on three dimensions:

1. **keywords:** Does the generated answer includes the expected keywords?



2. `correct_source`: Did the LLM use the correct source?
3. `llm_judge_passed`: Did the LLM-powered judge deem the answer semantically correct?

The LLM judge is a second instance of the same model (`llama3.2:3b`), instantiated independently via `get_judge_llm()`. It is prompted with a fixed evaluation instruction and constrained to return a binary decision, `true` or `false`, along with a single-sentence justification. Prior to evaluation, the judge undergoes a handshake test with a known prompt-response pair. If it fails to return `true`, the process aborts.

Initial test iterations used Ollama’s default generation parameters (`temperature = 0.8`, `top_p = 1.0`), which led to high variability in judgment outputs and inconsistent pass/fail classifications. Subsequent tuning reduced the judge temperature to 0.0 and set `repeat_penalty = 1.0`, which significantly improved determinism and alignment with the intended decision rule. The RAG model uses a slightly higher temperature (default: 0.2) to maintain fluency while remaining grounded in retrieved content.

Although both tasks require constrained generation, the judge’s role as a binary classifier demands near-deterministic behavior, whereas the RAG generator benefits from limited variability to support natural language fluency. Reliability of the judge was further improved through prompt engineering and failure-driven refinement of both the answer-generation and evaluation templates. Particular attention was given to avoiding false negatives arising from omission, paraphrasing, or defensible rewording. While the final pass/fail label is derived solely from the semantic judgment, keyword and source match are logged for interpretability and statistical purposes. This made designing and validating the approach easier.

## 6.2 Evaluation Results

To assess the system’s performance and consistency, a total of 240 test queries were executed across two hardware environments. Each query was tested in both `cold` and `warm` modes, yielding 60 queries per full evaluation run. On the desktop machine, three full test iterations were performed, resulting in 180 queries. On the laptop, a single iteration was executed, covering 60 queries. This made intra-system analysis (e.g., cold vs. warm behavior) and cross-system comparison (desktop vs. laptop) possible.

- **Laptop:** Intel i7 CPU with 8 logical cores, NVIDIA MX250 GPU (2GB VRAM)

## 6. EVALUATION

---

- **Desktop:** AMD Ryzen 9 3900X with 24 threads, NVIDIA GTX 1080 Ti (11GB VRAM)

All tests were executed using the same local models and retrieval pipeline. The RAG model used `llama3.2:3b` with `temperature = 0.2`, `top_p = 1.0`, and `repeat_penalty = 1.05`. The LLM judge was configured with `temperature = 0.0`, `top_p = 1.0`, and `repeat_penalty = 1.0`, to enforce deterministic binary evaluations.

### Pass Rate by Topic and Mode

Across all 180 desktop queries, overall pass rates were consistently high, with cold runs outperforming warm runs in each domain. Table 6.1 summarizes the number of passed and failed queries per topic and mode.

Topic	Mode	Total Tests	Passed	Pass rate (%)
Application Enrolment	Cold	30	27	90.00
Application Enrolment	Warm	30	23	76.67
Global	Cold	30	30	100.00
Global	Warm	30	29	96.67
Selection Placement	Cold	30	30	100.00
Selection Placement	Warm	30	25	83.33

**Table 6.1:** Pass rates desktop

Cold execution consistently yielded higher accuracy. The most pronounced gap was observed in the `application_enrolment` topic, where warm mode saw a 13.33% drop in pass rate. Manual inspection suggests that warm-mode prompt expansion sometimes led to context dilution, causing either retrieval imprecision or misalignment with expected answer focus.

The `global` and `selection_placement` domains showed near-perfect scores across all runs. In these cases, retrieved content was more self-contained and required less context continuity, reducing warm-mode sensitivity. The LLM judge exhibited consistent behavior across all test files, with no evidence of false negatives in the final tuned configuration.

### Latency Analysis

Table 6.2 reports the average latency metrics recorded on the desktop system. Each row reflects the mean of all queries for a given topic and mode, split into retrieval time, generation time, and total time. Standard deviation captures variance in total latency.

Topic	Mode	Total Time (s)	Retrieval (s)	Generation (s)
Application Enrolment	Cold	1.92	0.07	0.67
Application Enrolment	Warm	1.45	0.07	0.81
Global	Cold	1.78	0.07	0.57
Global	Warm	1.45	0.06	0.82
Selection Placement	Cold	1.78	0.07	0.54
Selection Placement	Warm	1.38	0.07	0.73

**Table 6.2:** Average latency by topic and execution mode on desktop (n=30 per row).

Total latency remained under 2 seconds in all cases. Cold runs incurred slightly higher overhead due to model instantiation on each query, while warm runs benefited from persistent model reuse. However, generation time was marginally higher in warm mode across all domains (likely due to longer prompts resulting from preserved chat history). This trade-off suggests that while warm mode reduces startup latency, it can incur additional token processing cost depending on prompt expansion.

Retrieval time remained stable across all configurations (0.06–0.07s), confirming that vector similarity search was not a performance bottleneck. The ChromaDB retriever scaled efficiently under both execution modes and document loads.

### Similarity Score Trends

Table 6.3 summarizes the average similarity scores of retrieved chunks per topic and mode. These scores reflect semantic proximity between the query embedding and the top-ranked document vectors retrieved by ChromaDB.

## 6. EVALUATION

---

Topic	Mode	Avg. Similarity Score
Application Enrolment	Cold	0.3429
Application Enrolment	Warm	0.3429
Global	Cold	0.2734
Global	Warm	0.2734
Selection Placement	Cold	0.2575
Selection Placement	Warm	0.2575

**Table 6.3:** Average similarity scores by topic and execution mode (desktop).

Warm and cold runs used identical query sets, and retrieval is independent of prompt history. As expected, similarity scores remained unchanged between modes, confirming that the retrieval pipeline operates deterministically given a fixed embedding model and query.

Queries under the `global` topic were constructed as a balanced mix—50% from `application_enrolment`, 50% from `selection_placement`. The lower average similarity score observed in this group likely reflects increased retrieval ambiguity due to topic blending. Nevertheless, the RAG system consistently selected top- $k$  candidates without score thresholding. Deduplication was applied post-retrieval, and the final prompt included up to two distinct chunks.

### RAG vs. Judge Contribution

The final test outcome (`PASS` or `FAIL`) reflects a compound result: the LLM must both generate an answer aligned with the reference and retrieve the correct source document. Internally, three criteria are evaluated per query:

- `matched_keywords` — lexical overlap with critical terms
- `matched_source` — correct document identified among retrieved chunks
- `llm_judge_passed` — binary semantic evaluation

Failures in earlier iterations were primarily due to the judge’s sensitivity to variation in formulation. Under default parameters (temperature = 0.8), the judge exhibited non-deterministic behavior and rejected correct responses due to minor paraphrasing or omitted

non-critical details. After tuning its temperature to 0.0 and refining the instruction prompt, the judge produced consistent decisions across all topics.

The dominant remaining failure cases were retrieval-related. In warm mode, prompt expansion sometimes diluted the query focus, resulting in a shift in retrieved context. This was most evident in the `application_enrolment` domain, where failure rate increased under warm conditions (Table 6.1). These errors were not caused by the judge, but by incorrect or insufficient evidence passed to the model.

Manual inspection of failure cases confirmed that the LLM judge operated deterministically under the final configuration (temperature = 0.0). Across three iterations of the desktop evaluation, pass/fail outcomes were consistent for nearly all queries. Where discrepancies did occur, they were attributed not to the judge, but to upstream limitations in retrieval or chunk scope.

In the `application_enrolment` cold runs, all failures occurred on the same query regarding diploma evaluation fees. The retrieved chunk truncated the relevant regulation, excluding critical details such as the full amount (€100) and the term “non-refundable.” The LLM hallucinated an incorrect fee (€50), and the judge correctly flagged the response as semantically incorrect. This points to chunk size as the limiting factor, not judgment error. In warm runs for the same topic, three queries failed despite the correct context being retrieved and correctly paraphrased by the model.

### Desktop vs. Laptop Comparison

Table 6.4 compares test results across the desktop and laptop systems. Each row summarizes one topic-mode combination. The same test set was used in both environments, with the laptop executing one full iteration (60 queries) and the desktop executing three (180 queries).

## 6. EVALUATION

System	Topic	Mode	Pass Rate (%)	Total Time (s)	Retrieval (s)	Generation (s)
Desktop	Application Enrolment	Cold	90.00	1.92	0.07	0.67
Desktop	Application Enrolment	Warm	76.67	1.45	0.07	0.81
Desktop	Global	Cold	100.00	1.78	0.07	0.57
Desktop	Global	Warm	96.67	1.45	0.06	0.82
Desktop	Selection Placement	Cold	100.00	1.78	0.07	0.54
Desktop	Selection Placement	Warm	83.33	1.38	0.07	0.73
Laptop	Application Enrolment	Cold	90.00	21.52	5.77	14.02
Laptop	Application Enrolment	Warm	90.00	28.95	6.53	21.17
Laptop	Global	Cold	100.00	21.11	6.29	13.25
Laptop	Global	Warm	90.00	25.46	6.3	17.86
Laptop	Selection Placement	Cold	100.00	23.10	6.34	14.27
Laptop	Selection Placement	Warm	80.00	23.21	6.37	15.63

**Table 6.4:** Cross-system comparison of evaluation metrics (n=10 per laptop row, n=30 per desktop row).

**Accuracy** Pass rates were broadly consistent across systems for most topic-mode pairs,

with all cold-mode tests scoring  $\geq 90\%$ . However, one significant divergence emerged: **Application Enrolment (Warm)** scored 76.67% on the desktop but 90.00% on the laptop (a 13.33 percentage point increase). The discrepancy likely reflects run-specific retrieval variation or more favorable chunk selection on the laptop. Aside from this case, other differences stayed within a  $\pm 3.33\%$  margin, suggesting stability across hardware.

**Latency.** Runtime differences were hardware-bound and substantial. On the desktop, end-to-end query time remained under 2 seconds for all cases. On the laptop, total time ranged from 21 to 29 seconds per query. The gap was consistent in both retrieval and generation stages: retrieval averaged  $\sim 0.07$ s on the desktop vs.  $\sim 6.3$ s on the laptop; generation ranged from  $\sim 0.5$ – $0.8$ s vs. 13–21s, respectively. These findings confirm that the MX250 was insufficient for interactive inference workloads at scale.

**Warm vs. Cold.** Warm mode consistently increased generation latency across both systems, as expected. Prompt expansion with retained history inflated decoding time. Accuracy effects were more variable: while warm mode slightly underperformed cold mode on the desktop in all three domains, the laptop showed no consistent drop, and even outperformed cold mode in **Application Enrolment**. This may reflect narrower history depth, fewer resource constraints during the run, or run-specific prompt shaping effects.

**Interpretation.** Overall, results confirm that the system executes identically across environments, with consistent pass/fail logic and retrieval behavior. However, throughput is tightly coupled to hardware acceleration, and warm-mode behavior remains more sensitive to retrieval variance. These constraints must be accounted for in deployments where latency or stability is critical.

## 6.3 Summary and Findings

The evaluation demonstrates that a local RAG system composed of open-source components can achieve high retrieval precision, low semantic error rates, and sub-second latency under suitable hardware conditions. Across 240 queries, executed in cold and warm modes over two hardware profiles, the pipeline showed strong reliability and predictable behavior. Most performance variation was attributable to controllable factors: retrieval quality, prompt length, and hardware acceleration.

Retrieval accuracy remained consistently high across all topics. The system selected semantically relevant chunks even when queries were phrased naturally and without structural cues. Similarity scores showed no divergence across cold and warm runs, validating the determinism of the embedding and vector search pipeline. Semantic correctness, as

## 6. EVALUATION

---

judged by a separate LLM instance, also stabilized after temperature tuning and prompt refinement. All observed test discrepancies could be traced to input truncation or vague completions, not failures in judgment consistency.

Cold runs consistently achieved higher pass rates than warm runs, especially in regulation-heavy domains such as `application_enrolment`. This was caused by chat history expansion, which occasionally diluted the focus of prompts and led to partial or hedged answers. Once warm-mode history was capped and chunk deduplication improved, performance converged across modes.

Latency analysis revealed sharp differences in throughput between systems. On the desktop, total query latency remained under 2 seconds in all configurations. On the laptop, queries ranged from 21 to 29 seconds, with both retrieval and generation bottlenecked by hardware. While both environments produced valid and reproducible outputs, only the desktop supported interactive use at scale.

Overall, the system fulfills its design goals: it operates entirely offline, responds deterministically under test, and enables fine-grained evaluation of LLM behavior via interpretable metrics. The results demonstrate that open-source LLMs can serve domain-specific assistant tasks effectively when paired with targeted retrieval and prompt control. In response to **RQ1.3**, the evaluation suggests that on-premise deployment becomes strategically viable when data sensitivity, domain specificity, or API latency constraints preclude external services. The feasibility of this approach scales with hardware. While the tested system achieved interactive latency on consumer-grade GPU hardware, latency remained prohibitive on older mobile GPUs. With dedicated inference accelerators or GPUs optimized for transformer models (e.g., high-core-count parallelization, large VRAM, low memory latency), the same pipeline could scale to larger models or higher throughput, further improving accuracy without sacrificing sovereignty. Thus, hardware quality (not just model size) defines the margin between usable and real-time open-source deployments in practice.



# 7

## Discussion

This chapter contextualizes the system design and evaluation results within the broader goals of the thesis. Each section corresponds to one of the research questions, examining how the implemented prototype answers them in practice. The chapter concludes with a discussion of limitations and directions for future work.

### 7.1 RQ1.1: Landscape of Open-Source LLMs and Tools

Open-source LLMs provide a deployment model in which inference, adaptation, and data handling remain under institutional control. As outlined in Chapter 2, such models eliminate dependency on external APIs and allow for full local execution. Minaee et al. [19] group open models into several families, LLaMA, GPT and PALM. These differ in architecture, tokenizer design, training data, and licensing. Most are released at multiple parameter scales and quantization levels, enabling deployment across a range of hardware configurations.

For this thesis, the model selected was `llama3.2:3b`, a compact and instruction-aligned variant from Meta’s LLaMA family. This choice was motivated in part by community guidance, specifically a walkthrough by Thomas Janssen [51], which demonstrated its balance between generation quality and hardware efficiency. The model’s compatibility with LangChain also made development much more effective. Although other models could have been evaluated, resource constraints and scope limitations made comparative testing infeasible. Nevertheless, the selected model proved effective for the target domain and hardware class.

### 7.2 RQ1.2: Domain Adaptation and Integration

Several approaches exist for adapting LLMs to specialized domains, ranging from full fine-tuning to lightweight parameter-efficient methods. However, this thesis only used RAG as the sole adaptation technique. RAG was chosen due to its lower computational footprint, its interpretability, and its ability to ground outputs in explicit textual sources. Unlike fine-tuning, RAG makes it straightforward to trace outputs to specific documents, which proves useful in a use case like the chatbot.

The implementation used ChromaDB as the vector store and used a dual-index retrieval setup: one global index and one per topic domain. Documents were chunked using a recursive splitter with a fixed window (600 tokens, 100 overlap). Several chunking configurations were tested during development, though the final parameters were constrained by prompt limits and generation stability.

Nonetheless, the chunking design posed limitations. Truncated or narrowly scoped chunks occasionally led to model hallucinations or incomplete answers. With access to more capable hardware and larger models, future deployments could increase chunk size and context depth, improving grounding and recall. More advanced adaptation methods were not explored here but remain promising extensions.

### 7.3 RQ1.3: Comparative Viability of On-Premise Deployment

This thesis set out to determine when and how open-source LLMs, deployed locally, offer strategic advantages over proprietary cloud-based solutions. The findings suggest that such deployments are feasible, performant, and offer distinct benefits, especially in contexts where control, traceability, or data privacy are critical.

Externally hosted LLMs introduce unavoidable data exposure. Each query must transit through third-party infrastructure. Furthermore, closed-source models cannot be inspected, modified, or optimized by the user. In contrast, the system developed here ran entirely on local hardware, under full control of the operator. All data, embeddings, and inference calls remained within the execution boundary, with no telemetry or usage restrictions.

From a performance standpoint, the system demonstrated near-instantaneous latency (<2s/query) on the desktop. Even on modest hardware (a laptop with an MX250 GPU), the same pipeline executed reliably, albeit with higher latency (25s/query). This validates

the blueprint’s flexibility: organizations can select models, optimization strategies, and compute targets based on their constraints. There is no vendor lock-in, no cost per query, and no restrictions on usage volume or customization.

However, the approach is not without trade-offs. There is no plug-and-play interface. Deployment requires familiarity with LLMs, vector databases, prompt design, and more. These costs are one-time but non-trivial. Still, for use cases where privacy and long-term autonomy are prioritized, the case for local, open-source deployment is technically and strategically sound.

## 7.4 Limitations and Future Work

**Synthetic Evaluation Data** Several constraints limited the scope of this thesis. First, test queries were manually authored using the embedded regulatory documents. While these questions reflect realistic phrasing and content, they do not originate from real user interactions. Incorporating authentic student inquiries would provide more representative evaluation data.

**LLM-Based Judgment** Second, although the LLM judge improved evaluation efficiency and reproducibility, it remains an LLM. Even at temperature = 0.0, it can produce inconsistent or brittle decisions in edge cases. Manual review of the 240 evaluation queries found no false positives or false negatives in this case. However, the broader interpretability challenge remains: generative evaluators lack transparent decision boundaries, and their outputs are not formally verifiable.

**Adaptation Strategy** Third, while the system combines RAG with constrained prompt design, this adaptation strategy remains limited in flexibility compared to approaches involving fine-tuning or tool-augmented reasoning. Larger models or fine-tuned adapters could support more nuanced answers or multi-document reasoning. Future research could test adapter-based fine-tuning, swap the embedding model, or experiment with long-context transformers capable of larger chunk sizes. Prompt construction in the current system prioritizes the top-ranked chunk for grounding and includes an additional chunk as secondary context. This structure performed reliably in testing, but prompting remains model-dependent, and no configuration generalizes across tasks. Further work on this should improve performance in other settings.

## 7. DISCUSSION

---

**Generality and Transferability** Finally, this thesis was intended as a blueprint. The architecture and evaluation logic are extensible to other domains, and any organization using commercial LLM APIs could in principle migrate to a private, on-premise alternative without compromising capability. With appropriate model selection and hardware provisioning, the techniques outlined here provide a scalable path toward transparent, private, and auditable LLM deployment.

**Environmental Scope** This thesis did not measure power consumption or carbon footprint. Nonetheless, the choice to run a 3B-parameter model locally—without reliance on cloud infrastructure or general-purpose LLMs—was intended to reduce environmental overhead. Larger commercial models incur energy and water costs both during training and inference [9], [11]. Future work could include empirical measurement of energy draw across deployment configurations and integrate ecological metrics into the assessment of LLM deployment strategies.

## Related Work

In this chapter, we review scientific and applied contributions that are comparable to this thesis in terms of goal, methodology, or system architecture. While some papers initially appear closely aligned, a more detailed examination reveals substantive differences in scope, implementation depth, or evaluation rigor. For these cases, a more thorough comparative analysis is provided to clarify where this work diverges and contributes beyond prior efforts.

### 8.1 Building UniGPT (Radas et al., 2023)

As an early example of institutional on-premise LLM deployment, Radas et al. (2023) [15] present UniGPT, a multilingual inference service hosted at the University of Münster. The system uses a ChatUI-based interface backed by a TGI inference server and quantized models such as `Mixtral 8x7B`. It is deployed on Kubernetes with Ceph storage and A40 GPUs, supporting concurrent users and dynamic model loading. The system focuses on inference-only access to multilingual open-weight models, selected primarily for German-language support. It integrates Shibboleth-based authentication and stores sessions in MongoDB. Although RAG is mentioned as a possible extension, no retrieval pipeline is implemented. Evaluation remains anecdotal, limited to early user feedback during a test phase. No task-specific adaptation, quantitative benchmarks, or traceable retrieval components are presented.

In contrast, as a part of this thesis, a domain-specific RAG assistant for university policy documents is developed, with all components (embedding, vector storage, generation) executed locally and without internet access. Unlike UniGPT, which operates as a general-purpose LLM gateway, this system performs grounded question answering through

## 8. RELATED WORK

---

a fully implemented semantic retrieval pipeline. The use case is narrow but operationalized end-to-end, including PDF ingestion, local embedding via `mxbai-embed-large`, and controlled prompt construction passed to a quantized 3B LLaMA model. The architecture is modular and hardware-flexible, with empirical evaluation conducted across both CPU-only and GPU-enabled environments to quantify latency variation under differing compute conditions. Evaluation includes structured test sets, LLM-as-judge validation, and latency decomposition across machines. Whereas UniGPT emphasizes infrastructure scaling, this work implements a task-specific pipeline optimized for privacy, traceability, and local resource limits.

### 8.2 Chat With OPD (Bhise, 2024)

Bhise (2024) [14] explores a goal closely aligned with this thesis: the development of a localized, privacy-preserving RAG system deployed fully on-premise. The system discusses a GUI module, preprocessing logic for PDF ingestion and chunking, a FAISS-based vector store, and a local LLM for embedding generation and response synthesis. The design prioritizes offline execution and user privacy, with all components assumed to run on local infrastructure. FAISS is chosen for its lightweight footprint among open-source vector databases, and the system is modularized into preprocessing, embedding, and query handling stages. While the pipeline architecture is clearly articulated, no specific language model or quantization configuration is described, and there is no discussion of how retrieval accuracy is managed or how the system behaves under hardware constraints. The project frames itself as a customizable template for privacy-focused RAG deployment, particularly in sensitive domains such as healthcare and legal services, but remains conceptual in its evaluation.

This thesis differs from Bhise’s in several substantive ways. Most notably, it develops and evaluates a task-specific RAG system grounded in structured regulatory documents from a university domain. All components—embedding via `‘mxbai-embed-large’`, retrieval via `ChromaDB`, generation via a quantized 3B LLaMA model—are executed locally, with model selection and latency trade-offs empirically benchmarked across CPU and GPU environments. Unlike *Chat with OPD*, this system defines a full ingestion-to-evaluation loop, including test sets, hardware-specific profiling, and LLM-as-judge validation for semantic correctness. It also incorporates fallback logic, prompt design constraints, and session memory to manage conversation flow. Where Bhise identifies challenges such as chunk calibration and content omission as future work, this thesis integrates those concerns into its implementation and evaluation design. The resulting system is not only functionally

complete but serves as a reproducible reference architecture for domain-specific, privacy-preserving LLM deployment.

### 8.3 UvA AI Chat (Pepijn Stoop, 2025)

Although not a scientific publication, the UvA AI Chat initiative is described in a 2025 article by Folia, the independent journalistic platform of the University of Amsterdam [16]. The piece reports on a series of institutional pilots in which a customized version of ChatGPT, termed UvA AI Chat, was deployed across faculties to support students and teachers in academic tasks. The system introduces a set of role-specific chat agents embedded within a UvA-controlled interface, designed to assist with writing, feedback, and curriculum support. While user-facing components are managed internally, the underlying language model remains external: OpenAI’s ChatGPT, accessed via Microsoft Azure. The system attempts to mitigate privacy risks by anonymizing inputs and purging user data post-inference, but nonetheless relies on cloud-hosted infrastructure and proprietary models. Pilot results indicated positive student reception, especially for writing tasks, though some educators questioned the reliability and pedagogical soundness of outputs..

This thesis differs from UvA AI Chat in both implementation and scope. Whereas the UvA system layers institutional access control over a closed-source backend, the present work operates entirely on-premise with open-weight models, local vector storage, and document-grounded prompting. UvA AI Chat offers no retrieval capabilities and is not domain-adapted. The result is a reproducible reference implementation for institutions seeking both access control over AI tools and functional sovereignty over model behavior, data exposure, and task relevance.

### 8.4 On-Premises Knowledge Repository (Dobur et al., 2024)

Dobur et al. (2024) [56] propose a Live Knowledge Library (LKL): an on-premise, enterprise-scale architecture that integrates internal data sources with fine-tuned LLMs to support real-time, context-aware information access. The system processes both structured and unstructured organizational data, applies supervised fine-tuning and reinforcement learning from human feedback, and delivers responses through a natural language interface layered over semantic search. The infrastructure is containerized and locally hosted, with a focus on compliance, system control, and user experience.

## 8. RELATED WORK

---

While the LKL shares this thesis’s emphasis on private, on-premise deployment, the two systems diverge sharply in purpose and method. Dobur et al. frame their solution as a general-purpose enterprise knowledge layer, whereas this thesis implements a task-specific, document-grounded assistant for structured QA. RAG is not used in LKL; responses are generated from a fine-tuned model rather than dynamically retrieved context. The architecture assumes continued access to organizational training data and compute for iterative refinement. In contrast, this work targets low-overhead deployment using static policy documents and avoids fine-tuning entirely. Evaluation in Dobur et al. is limited to qualitative insights, whereas this thesis presents empirical tests of retrieval accuracy, semantic relevance, and latency under constrained hardware conditions.

### 8.5 Towards On-Premise Hosted LLM (Hedlund, 2024)

Hedlund (2024) [57] explores how open-source LLMs can be used on-premise to generate documentation for proprietary codebases in privacy-sensitive settings. Two architectures are compared: one where the LLM generates summaries directly from source code, and another that appends retrieved documentation to the prompt—an implicit form of RAG. Models such as Mixtral and Code LLaMA are deployed locally on GPU servers, and output quality is evaluated through a mix of multiple-choice assessments and developer feedback surveys.

The overlap with this thesis lies in the shared focus on local inference and constrained environments. However, Hedlund’s system is narrowly tailored to code summarization and does not implement persistent vector-based retrieval or any modular RAG infrastructure. The retrieval stage is tightly coupled to prompt construction and lacks a separate embedding layer or index. Unlike this work, which targets a document-grounded assistant built for QA over regulatory texts, Hedlund emphasizes generation fidelity without addressing retrieval accuracy, latency decomposition, or architectural generalizability. Where Hedlund tests usability for developers, this thesis evaluates system behavior under varied hardware constraints and tracks semantic alignment through structured test sets and LLM-based scoring.

### 8.6 On-Site Deployment (Schillaci, 2024)

Schillaci (2024) [58] outlines the strategic case for deploying LLMs entirely on-site, especially in security-sensitive sectors where regulatory and data exposure risks make cloud-



based AI untenable. The work surveys available tooling—such as vLLM, FastChat, and model quantization techniques like GPTQ—and discusses practical considerations around hardware (e.g., consumer GPUs), energy usage, and threat modeling. The emphasis is on framing self-hosted inference as a matter of cybersecurity posture, not just infrastructure choice. Unlike this thesis, the chapter does not present an implemented system or any empirical evaluation. It assumes high-end infrastructure and targets multi-user production settings, focusing on general deployment considerations rather than retrieval architecture or domain adaptation. RAG methods, evaluation design, and system modularity are outside its scope. Where this thesis defines and tests a constrained, document-grounded system under hardware limits, Schillaci’s contribution remains a high-level framing of infrastructural risk and readiness.

## 8.7 MedAide (Basit et al., 2024)

Basit et al. (2024) [59] present MedAide, a lightweight, domain-specific assistant for medical diagnostics that shares this thesis’s emphasis on edge deployment and offline inference. The system deploys quantized open-weight LLMs—optimized with LoRA and Q4/Q8 precision—on edge devices such as Nvidia Jetson boards. A LangChain-integrated retrieval component supports prescription lookup via FAISS, with hallucination mitigation as a core goal. Evaluation includes domain-specific benchmarks (e.g., USMLE-style questions) and human assessments using GPT-4. LLaMA 2 7B is selected for its balance of accuracy and resource efficiency.

Unlike this thesis, MedAide targets a single high-stakes vertical—preliminary medical diagnostics—while relying on multiple model architectures and aggressive fine-tuning. The retrieval component uses similarity search over preprocessed medical texts but does not formalize RAG evaluation or document grounding. Evaluation lacks prompt transparency, latency breakdown, or modular ablation. While the system excels in edge-device feasibility, its design does not generalize to broader document-based institutional QA settings. The present thesis focuses on reproducible, domain-agnostic RAG infrastructure and systematic validation under constrained compute environments.

### 8.8 Summary and Alignment with Research Questions

#### 8.8.1 Honorable Mentions and Implementation Influences

While this chapter has focused on peer-reviewed literature and formally evaluated systems, several community-driven, open-source resources have influenced the implementation of this thesis. In particular, Dutch software developer Thomas Janssen has produced YouTube tutorials and GitHub repositories demonstrating how to implement local RAG systems using Python, LangChain, Streamlit or Gradio, and ChromaDB [49], [50]. These implementations span both PDF-based ingestion pipelines using OpenAI models and agentic, locally hosted pipelines relying on Ollama and web-scraped Wikipedia content [51]–[53]. Part of the ingestion logic developed in this thesis derives from Janssen’s publicly available example code [60], particularly the structure for document loading, chunking, and vector ingestion. This codebase served as an initial scaffold, later adapted for structured university documents, modified chunk sizes, and integration into a hardware-flexible pipeline. While not part of the peer-reviewed corpus, such practitioner contributions illustrate the evolving ecosystem of open-source experimentation and provided a practical baseline during early development. Additional frameworks, such as LangChain and Ollama, are also referenced in Janssen’s videos and were selected for this thesis due to their support for local model orchestration and vector store integration.

#### 8.8.2 Research Gap and Contribution

This chapter has surveyed a diverse set of systems and publications that explore on-premise LLM deployment, local RAG architecture, and privacy-preserving adaptation of language models. While no reviewed work replicates the specific design constraints and evaluation scope of this thesis, the comparative analysis clarifies how this implementation builds upon, diverges from, or operationalizes aspects of prior contributions.

Several works (e.g., [15], [14], [16]) approach institutional use of LLMs through either interface customization or local deployment, but stop short of implementing or benchmarking a fully realized RAG pipeline. Others (e.g., [56], [57]) explore domain-specific adaptation and internal knowledge access, yet often rely on fine-tuning rather than retrieval-based grounding. In contrast, this thesis defines and evaluates a full-stack RAG architecture implemented entirely on-premise, with quantized models, local embedding, and end-to-end test coverage.

This thesis contributes a concrete reference implementation that integrates all core components of a local RAG system: document ingestion, embedding, retrieval, generation, and

## 8.8 Summary and Alignment with Research Questions

---

evaluation, under privacy and hardware constraints. The reviewed literature demonstrates the relevance and growing interest in on-premise LLM workflows, but reveals a lack of end-to-end systems that combine functional grounding, empirical evaluation, and strict local execution. The system developed here addresses that gap by operationalizing a domain-specific assistant with reproducible, modular components and transparent performance characterization.

## 8. RELATED WORK

---

# Conclusion

## 9.1 Summary of Approach and Findings

This thesis set out to answer the main research question, as noted in the introduction:

*How can organizations adopt and adapt open-source LLMs on-premise to achieve data sovereignty and domain-specific performance, while overcoming the limitations of closed-source alternatives?*

To approach this question, we first examined the current landscape of open-source language models and supporting tools suitable for self-hosted deployment. We also looked at the trade-offs between open and closed-source models, particularly in relation to transparency, privacy, and operational control. Methods for domain adaptation were discussed through a focused review of both training-time and inference-time techniques. These findings, covered in the background chapter, informed the design principles of the system developed in this thesis.

Based on these insights, we implemented and evaluated a fully local assistant system using a retrieval-augmented generation pipeline. The system operates entirely offline, using institutional documents, local embeddings, and a compact open-source language model served through Ollama. Evaluation across two hardware setups showed strong accuracy, consistent retrieval quality, and predictable performance under resource constraints. While the implementation ran on opportunistic consumer hardware, the system is designed to benefit from larger models and stronger infrastructure. Latency decreased significantly on a more capable machine, confirming that on-premise LLMs can achieve interactive performance under the right conditions. The system remains a blueprint: modular, reproducible, and adaptable to other institutional domains.

## 9. CONCLUSION

---

### 9.2 Answering the Research Questions

The main research question is supported by three sub-questions, each addressed in a different phase of the work.

- **RQ1: Model Landscape and Tools.** This question is addressed in the background chapter, where we reviewed the architecture, scale, licensing, and ecosystem support of major open-source LLMs. This provided the basis for selecting a suitable model for on-premise use, balancing performance with hardware feasibility.
- **RQ2: Domain Adaptation and Integration.** This is addressed both theoretically and practically. The background chapter discusses available methods, including fine-tuning, parameter-efficient tuning, and RAG. In the system design and implementation, we apply a RAG-based approach, which enables integration of internal documents without modifying model weights or exposing data.
- **RQ3: Comparative Viability.** This is addressed through the evaluation and discussion. The results show that small, locally hosted models paired with efficient retrieval and targeted prompts, can offer competitive performance. When privacy, data locality, or deployment control are prioritized, open-source, on-premise LLMs present a credible and in many cases superior alternative to externally hosted models.

Together, these results support the main hypothesis: that open-source LLMs can be effectively deployed and adapted within an organization, even under limited resources, to achieve domain-specific functionality without external dependencies.

### 9.3 Outlook and Future Work

While the system performs well within its design constraints, it remains a starting point. Future work could extend the adaptation strategy beyond RAG by incorporating fine-tuning or adapter-based methods. Evaluating the system with real user queries would allow for more representative assessment and better insight into failure cases. More capable hardware would enable support for larger models and longer contexts, improving grounding and response quality.

The system architecture is general enough to be reused in other domains. With minimal changes, the blueprint could be applied to regulated fields such as healthcare, legal services, or public administration. As models and tooling continue to improve, the case for private,

### **9.3 Outlook and Future Work**

---

transparent, and task-specific LLM deployments is likely to strengthen. This thesis offers one practical path toward that goal.

## 9. CONCLUSION

---



# Bibliography

- [1] “AI | 2024 Stack Overflow Developer Survey.” (n.d.), [Online]. Available: <https://survey.stackoverflow.co/2024/ai/> (visited on 04/06/2025).
- [2] Springs. “Large Language Model Statistics And Numbers (2025) - Springs.” (), [Online]. Available: <https://springsapps.com/knowledge/large-language-model-statistics-and-numbers-2024,%20https://springsapps.ai/blog/large-language-model-statistics-and-numbers-2024> (visited on 04/06/2025).
- [3] “Data usage for consumer services FAQ | OpenAI Help Center.” (), [Online]. Available: <https://help.openai.com/en/articles/7039943-data-usage-for-consumer-services-faq> (visited on 03/26/2025).
- [4] OpenAI. “How your data is used to improve model performance | OpenAI Help Center.” (), [Online]. Available: <https://help.openai.com/en/articles/5722486-how-your-data-is-used-to-improve-model-performance> (visited on 03/26/2025).
- [5] “AI Act | Shaping Europe’s digital future.” (Mar. 20, 2025), [Online]. Available: <https://digital-strategy.ec.europa.eu/en/policies/regulatory-framework-ai> (visited on 03/26/2025).
- [6] European Parliament. Directorate General for Parliamentary Research Services., *The Impact of the General Data Protection Regulation on Artificial Intelligence*. LU: Publications Office, 2020. [Online]. Available: <https://data.europa.eu/doi/10.2861/293> (visited on 03/26/2025).
- [7] T. Mayover. “When AI Technology and HIPAA Collide,” *The HIPAA Journal*. (Oct. 2, 2024), [Online]. Available: <https://www.hipaajournal.com/when-ai-technology-and-hipaa-collide/> (visited on 03/26/2025).
- [8] M. Mäntymäki, M. Minkkinen, T. Birkstedt, *et al.* “Putting AI Ethics into Practice: The Hourglass Model of Organizational AI Governance.” arXiv: 2206.00335 [cs]. (Jan. 31, 2023), [Online]. Available: <http://arxiv.org/abs/2206.00335> (visited on 03/26/2025), pre-published.

## BIBLIOGRAPHY

---

- [9] “Explained: Generative AI’s environmental impact,” MIT News | Massachusetts Institute of Technology. (Jan. 17, 2025), [Online]. Available: <https://news.mit.edu/2025/explained-generative-ai-environmental-impact-0117> (visited on 07/01/2025).
- [10] S. Deb, “Saying ‘Thank You’ to ChatGPT Is Costly. But Maybe It’s Worth the Price.,” *The New York Times Technology*, Apr. 24, 2025, ISSN: 0362-4331. [Online]. Available: <https://www.nytimes.com/2025/04/24/technology/chatgpt-alexaplease-thank-you.html> (visited on 07/01/2025).
- [11] P. Li, J. Yang, M. A. Islam, *et al.* “Making AI Less “Thirsty”: Uncovering and Addressing the Secret Water Footprint of AI Models.” arXiv: 2304.03271 [cs]. (Mar. 26, 2025), [Online]. Available: <http://arxiv.org/abs/2304.03271> (visited on 07/01/2025), pre-published.
- [12] S. Ren, B. Tomlinson, R. W. Black, *et al.*, “Reconciling the contrasting narratives on the environmental impact of large language models,” *Scientific Reports*, vol. 14, no. 1, p. 26310, Nov. 1, 2024, ISSN: 2045-2322. DOI: 10.1038/s41598-024-76682-6. [Online]. Available: <https://www.nature.com/articles/s41598-024-76682-6> (visited on 07/01/2025).
- [13] EIOPA. “Open-source tools for the modelling and management of climate change risks - EIOPA,” Open-source tools for the modelling and management of climate change risks. (), [Online]. Available: [https://www.eiopa.europa.eu/tools-and-data/open-source-tools-modelling-and-management-climate-change-risks\\_en](https://www.eiopa.europa.eu/tools-and-data/open-source-tools-modelling-and-management-climate-change-risks_en) (visited on 07/01/2025).
- [14] G. Bhise. “Chat with OPD (On-Premise-Data),” Chat with OPD (On-Premise-Data). (2024), [Online]. Available: <https://scholarworks.calstate.edu/concern/projects/9g54xs77c> (visited on 03/17/2025).
- [15] J. Radas, B. Risse, and R. Vogl, “Building UniGPT: A Customizable On-Premise LLM-Solution for Universities,” presented at the Proceedings of EUNIS 2024 Annual Congress in Athens, pp. 108–98. DOI: 10.29007/jv11. [Online]. Available: <https://easychair.org/publications/paper/CDHx> (visited on 03/17/2025).
- [16] “UvA created its own ChatGPT for students and teachers. Is it safer than the original?” (), [Online]. Available: <https://www.folia.nl/en/actueel/164740/uva-created-its-own-chatgpt-for-students-and-teachers-is-it-safer-than-the-original> (visited on 04/06/2025).
- [17] J. Devlin, M.-W. Chang, K. Lee, *et al.* “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” arXiv: 1810.04805 [cs]. (May 24,

- 2019), [Online]. Available: <http://arxiv.org/abs/1810.04805> (visited on 05/16/2025), pre-published.
- [18] Y. Chang, X. Wang, J. Wang, *et al.*, “A Survey on Evaluation of Large Language Models,” *ACM Trans. Intell. Syst. Technol.*, vol. 15, no. 3, 39:1–39:45, Mar. 29, 2024, ISSN: 2157-6904. DOI: 10.1145/3641289. [Online]. Available: <https://dl.acm.org/doi/10.1145/3641289> (visited on 05/15/2025).
- [19] S. Minaee, T. Mikolov, N. Nikzad, *et al.* “Large Language Models: A Survey.” arXiv: 2402.06196 [cs]. (Mar. 23, 2025), [Online]. Available: <http://arxiv.org/abs/2402.06196> (visited on 05/15/2025), pre-published.
- [20] “An empirical study of smoothing techniques for language modeling,” *Computer Speech & Language*, vol. 13, no. 4, pp. 359–394, Oct. 1, 1999, ISSN: 0885-2308. DOI: 10.1006/csla.1999.0128. [Online]. Available: <https://www.sciencedirect.com/science/article/abs/pii/S088523089901286> (visited on 06/16/2025).
- [21] Y. Bengio, R. Ducharme, P. Vincent, *et al.*, “A Neural Probabilistic Language Model,”
- [22] P.-S. Huang, X. He, J. Gao, *et al.*, “Learning deep structured semantic models for web search using clickthrough data,” in *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, ser. CIKM ’13, New York, NY, USA: Association for Computing Machinery, Oct. 27, 2013, pp. 2333–2338, ISBN: 978-1-4503-2263-8. DOI: 10.1145/2505515.2505665. [Online]. Available: <https://doi.org/10.1145/2505515.2505665> (visited on 06/16/2025).
- [23] I. Sutskever, O. Vinyals, and Q. V. Le. “Sequence to Sequence Learning with Neural Networks.” arXiv: 1409.3215 [cs]. (Dec. 14, 2014), [Online]. Available: <http://arxiv.org/abs/1409.3215> (visited on 06/16/2025), pre-published.
- [24] J. Devlin, M.-W. Chang, K. Lee, *et al.* “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” arXiv: 1810.04805 [cs]. (May 24, 2019), [Online]. Available: <http://arxiv.org/abs/1810.04805> (visited on 06/16/2025), pre-published.
- [25] J. Kaplan, S. McCandlish, T. Henighan, *et al.* “Scaling Laws for Neural Language Models.” arXiv: 2001.08361 [cs]. (Jan. 23, 2020), [Online]. Available: <http://arxiv.org/abs/2001.08361> (visited on 06/16/2025), pre-published.
- [26] A. Chowdhery, S. Narang, J. Devlin, *et al.* “PaLM: Scaling Language Modeling with Pathways.” arXiv: 2204.02311 [cs]. (Oct. 5, 2022), [Online]. Available: <http://arxiv.org/abs/2204.02311> (visited on 06/16/2025), pre-published.
- [27] A. Vaswani, N. Shazeer, N. Parmar, *et al.* “Attention Is All You Need.” arXiv: 1706.03762 [cs]. (Aug. 2, 2023), [Online]. Available: <http://arxiv.org/abs/1706.03762> (visited on 05/16/2025), pre-published.

## BIBLIOGRAPHY

---

- [28] H. Touvron, T. Lavril, G. Izacard, *et al.* “LLaMA: Open and Efficient Foundation Language Models.” arXiv: 2302.13971 [cs]. (Feb. 27, 2023), [Online]. Available: <http://arxiv.org/abs/2302.13971> (visited on 06/16/2025), pre-published.
- [29] T. B. Brown, B. Mann, N. Ryder, *et al.* “Language Models are Few-Shot Learners.” arXiv: 2005.14165 [cs]. (Jul. 22, 2020), [Online]. Available: <http://arxiv.org/abs/2005.14165> (visited on 06/16/2025), pre-published.
- [30] L. Ouyang, J. Wu, X. Jiang, *et al.* “Training language models to follow instructions with human feedback.” arXiv: 2203.02155 [cs]. (Mar. 4, 2022), [Online]. Available: <http://arxiv.org/abs/2203.02155> (visited on 06/16/2025), pre-published.
- [31] H. W. Chung, L. Hou, S. Longpre, *et al.* “Scaling Instruction-Finetuned Language Models.” arXiv: 2210.11416 [cs]. (Dec. 6, 2022), [Online]. Available: <http://arxiv.org/abs/2210.11416> (visited on 06/16/2025), pre-published.
- [32] J. Wei, X. Wang, D. Schuurmans, *et al.*, “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models,”
- [33] OpenAI. “GPT-4 Technical Report.” version 3. arXiv: 2303.08774 [cs]. (Mar. 27, 2023), [Online]. Available: <http://arxiv.org/abs/2303.08774> (visited on 06/16/2025), pre-published.
- [34] R. Anil, A. M. Dai, O. Firat, *et al.* “PaLM 2 Technical Report.” arXiv: 2305.10403 [cs]. (Sep. 13, 2023), [Online]. Available: <http://arxiv.org/abs/2305.10403> (visited on 06/16/2025), pre-published.
- [35] M. Chen, J. Tworek, H. Jun, *et al.* “Evaluating Large Language Models Trained on Code.” arXiv: 2107.03374 [cs]. (Jul. 14, 2021), [Online]. Available: <http://arxiv.org/abs/2107.03374> (visited on 06/16/2025), pre-published.
- [36] R. Nakano, J. Hilton, S. Balaji, *et al.* “WebGPT: Browser-assisted question-answering with human feedback.” arXiv: 2112.09332 [cs]. (Jun. 1, 2022), [Online]. Available: <http://arxiv.org/abs/2112.09332> (visited on 06/16/2025), pre-published.
- [37] “Tools models · Ollama Search.” (), [Online]. Available: <https://ollama.com/search> (visited on 06/16/2025).
- [38] “Stanford CRFM.” (), [Online]. Available: <https://crfm.stanford.edu/2023/03/13/alpaca.html> (visited on 06/16/2025).
- [39] T. Dettmers, A. Pagnoni, A. Holtzman, *et al.* “QLoRA: Efficient Finetuning of Quantized LLMs.” arXiv: 2305.14314 [cs]. (May 23, 2023), [Online]. Available: <http://arxiv.org/abs/2305.14314> (visited on 06/16/2025), pre-published.
- [40] “Koala: A Dialogue Model for Academic Research – The Berkeley Artificial Intelligence Research Blog.” (), [Online]. Available: <https://bair.berkeley.edu/blog/2023/04/03/koala/> (visited on 06/16/2025).

## BIBLIOGRAPHY

---

- [41] A. Q. Jiang, A. Sablayrolles, A. Mensch, *et al.* “Mistral 7B.” arXiv: 2310.06825 [cs]. (Oct. 10, 2023), [Online]. Available: <http://arxiv.org/abs/2310.06825> (visited on 06/16/2025), pre-published.
- [42] “Open-Source LLMs vs Closed: Unbiased Guide for Innovative Companies [2025].” (), [Online]. Available: <https://hatchworks.com/blog/gen-ai/open-source-vs-closed-llms-guide/> (visited on 06/17/2025).
- [43] “Open Source vs. Closed Source in Language Models: Pros and Cons - DS Stream Blog.” (), [Online]. Available: <https://www.dsstream.com/post/open-source-vs-closed-source-in-language-models-pros-and-cons> (visited on 06/17/2025).
- [44] O.-O. D. Science. “The Benefits of Open-Source vs. Closed-Source LLMs,” Medium. (Jan. 10, 2025), [Online]. Available: <https://odsc.medium.com/the-benefits-of-open-source-vs-closed-source-llms-71201e049bc7> (visited on 06/17/2025).
- [45] V. Hanke, T. Blanchard, F. Boenisch, *et al.*, “Open LLMs are Necessary for Current Private Adaptations and Outperform their Closed Alternatives,” *Advances in Neural Information Processing Systems*, vol. 37, pp. 1220–1250, Dec. 16, 2024. [Online]. Available: [https://proceedings.neurips.cc/paper\\_files/paper/2024/hash/02802e3df178cce7b13e8f63dd29ad9f-Abstract-Conference.html](https://proceedings.neurips.cc/paper_files/paper/2024/hash/02802e3df178cce7b13e8f63dd29ad9f-Abstract-Conference.html) (visited on 05/16/2025).
- [46] Y. Gao, Y. Xiong, X. Gao, *et al.* “Retrieval-Augmented Generation for Large Language Models: A Survey.” arXiv: 2312.10997 [cs]. (Mar. 27, 2024), [Online]. Available: <http://arxiv.org/abs/2312.10997> (visited on 07/01/2025), pre-published.
- [47] LangChain. “Build a Retrieval Augmented Generation (RAG) App: Part 1,” Build a Retrieval Augmented Generation (RAG) App: Part 1. (), [Online]. Available: <https://python.langchain.com/docs/tutorials/rag/> (visited on 06/23/2025).
- [48] B. Peng, M. Galley, P. He, *et al.* “Check Your Facts and Try Again: Improving Large Language Models with External Knowledge and Automated Feedback.” arXiv: 2302.12813 [cs]. (Mar. 8, 2023), [Online]. Available: <http://arxiv.org/abs/2302.12813> (visited on 07/01/2025), pre-published.
- [49] “Thomas Janssen,” YouTube. (), [Online]. Available: [https://www.youtube.com/channel/UCstCuI\\_7DvHw7-KSA3Yy9PQ](https://www.youtube.com/channel/UCstCuI_7DvHw7-KSA3Yy9PQ) (visited on 06/24/2025).
- [50] “ThomasJanssen-tech (Thomas Janssen).” (), [Online]. Available: <https://github.com/ThomasJanssen-tech> (visited on 06/24/2025).
- [51] Thomas Janssen, director, *Finally a Local RAG That WORKS!! (+ FULL RAG Pipeline)*, May 27, 2025. [Online]. Available: <https://www.youtube.com/watch?v=c5jHhMXmXyo> (visited on 06/24/2025).

## BIBLIOGRAPHY

---

- [52] “ThomasJanssen-tech/Local-RAG-with-Ollama: Build a 100% local Retrieval Augmented Generation (RAG) system with Python, LangChain, Ollama and ChromaDB!” (), [Online]. Available: <https://github.com/ThomasJanssen-tech/Local-RAG-with-Ollama/tree/main> (visited on 06/24/2025).
- [53] Thomas Janssen, director, *Build a Chatbot with RAG in 10 minutes / Python, LangChain, OpenAI*, Jan. 8, 2025. [Online]. Available: <https://www.youtube.com/watch?v=xf3gAFclwqo> (visited on 06/24/2025).
- [54] “General · victorwie/uni-chat,” GitHub. (), [Online]. Available: <https://github.com/victorwie/uni-chat> (visited on 07/01/2025).
- [55] “CUDA Toolkit - Free Tools and Training,” NVIDIA Developer. (), [Online]. Available: <https://developer.nvidia.com/cuda-toolkit> (visited on 06/29/2025).
- [56] B. Dobur, E. Bıçakçı, A. Terim, *et al.*, “The Building an On-Premises Knowledge Repository with Large Language Models for Instant Information Access,” *Orclever Proceedings of Research and Development*, vol. 5, no. 1, pp. 261–273, 1 Dec. 31, 2024, ISSN: 2980-020X. DOI: 10.56038/oprd.v5i1.545. [Online]. Available: <https://www.journals.orclever.com/oprd/article/view/545> (visited on 03/19/2025).
- [57] L. Hedlund, *Towards On-Premise Hosted Language Models for Generating Documentation in Programming Projects*. 2024. [Online]. Available: <https://urn.kb.se/resolve?urn=urn:nbn:se:ltu:diva-107590> (visited on 03/17/2025).
- [58] “(PDF) On-Site Deployment of LLMs,” in *ResearchGate*. DOI: 10.1007/978-3-031-54827-7\_23. [Online]. Available: [https://www.researchgate.net/publication/381092812\\_On-Site\\_Deployment\\_of\\_LLMs](https://www.researchgate.net/publication/381092812_On-Site_Deployment_of_LLMs) (visited on 03/17/2025).
- [59] A. Basit, K. Hussain, M. A. Hanif, *et al.* “MedAide: Leveraging Large Language Models for On-Premise Medical Assistance on Edge Devices.” arXiv: 2403.00830 [cs]. (Feb. 28, 2024), [Online]. Available: <http://arxiv.org/abs/2403.00830> (visited on 03/19/2025), pre-published.
- [60] “Chatbot-with-RAG-and-LangChain/ingest\_database.py at main · ThomasJanssen-tech/Chatbot-with-RAG-and-LangChain.” (), [Online]. Available: [https://github.com/ThomasJanssen-tech/Chatbot-with-RAG-and-LangChain/blob/main/ingest\\_database.py](https://github.com/ThomasJanssen-tech/Chatbot-with-RAG-and-LangChain/blob/main/ingest_database.py) (visited on 06/29/2025).

# Appendix

The full implementation developed for this thesis is available as an open-source repository at:

`https://github.com/victorwie/uni-chat`

The repository includes all code necessary to reproduce the system described in this thesis. Including:

- Data ingestion and preprocessing pipeline
- Retrieval-augmented generation engine
- Local inference setup using Ollama
- Evaluation scripts and test sets
- Streamlit-based user interface

Instructions for installation, configuration, and usage are provided in the README file.