



UNIVERSITY OF PIRAEUS - DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc “Advanced Informatics and Computing Systems – Software Development and Artificial Intelligence”

ΠΜΣ «Προηγμένα Συστήματα Πληροφορικής – Ανάπτυξη Λογισμικού και Τεχνητής Νοημοσύνης»

MSc Thesis

Μεταπτυχιακή Διατριβή

Thesis Title: Τίτλος Διατριβής:	RAG-Based Architecture Enhanced Specialized Chatbot Generation Δημιουργία Εξειδικευμένου Chatbot βασισμένο στην Αρχιτεκτονική RAG
Student's name-surname: Όνοματεπώνυμο φοιτητή:	Ioannis Ntourmas Ιωάννης Ντούρμας
Father's name: Πατρώνυμο:	Evangelos Ευάγγελος
Student's ID No: Αριθμός Μητρώου:	MPSP2327 ΜΠΣΠ2327
Supervisor: Επιβλέπων:	Dionysios Sotiropoulos, Assistant Professor Διονύσιος Σωτηρόπουλος, Επίκουρος Καθηγητής

March 2025 / Μάρτιος 2025

3-Member Examination Committee

Τριμελής Εξεταστική Επιτροπή

D. Sotiropoulos

Assistant Professor

Διονύσιος Σωτηρόπουλος
Επίκουρος Καθηγητής

G. Tsihrintzis

Professor

Γεώργιος Τσιχριντζής
Καθηγητής

E. Sakkopoulos

Associate Professor

Ευάγγελος Σακκόπουλος
Αναπληρωτής Καθηγητής

Abstract

The growing need for specialized chatbots, tailored to distinct domains, is increasingly evident across a wide range of industries and academic institutions. This thesis explores the design and implementation of a chatbot grounded in a Retrieval-Augmented Generation (RAG) architecture, specifically developed for the University of Piraeus. By integrating GPT-3.5 with a data retrieval system based on MongoDB, the chatbot delivers accurate and context-relevant answers for university-related queries. The dissertation focuses on the design principles employed in Flask for backend development, the React-based frontend for a user-friendly interface, and the optimization strategies that facilitate seamless data management. Experimental evaluations confirm the chatbot's effectiveness in providing domain-specific answers, underscoring its flexibility and reliability for academic needs.

Περίληψη

Η αυξανόμενη ανάγκη για εξειδικευμένα chatbots, προσαρμοσμένα σε συγκεκριμένους τομείς, γίνεται ιδιαίτερα εμφανής σε διάφορους κλάδους και ακαδημαϊκά ιδρύματα. Η παρούσα διπλωματική εργασία εξετάζει τον σχεδιασμό και την υλοποίηση ενός chatbot βασισμένου στην αρχιτεκτονική Retrieval-Augmented Generation (RAG), ειδικά προσαρμοσμένου για το Πανεπιστήμιο Πειραιώς. Με τη χρήση του GPT-3.5 σε συνδυασμό με ένα σύστημα ανάκτησης δεδομένων που αξιοποιεί τη MongoDB, το chatbot παρέχει ακριβείς και συναφείς απαντήσεις σε ζητήματα που αφορούν το πανεπιστήμιο. Η εργασία εστιάζει στις αρχές σχεδιασμού που εφαρμόζονται στο Flask για την ανάπτυξη του backend, στο React-based frontend για τη δημιουργία ενός φιλικού περιβάλλοντος χρήσης, καθώς και στις στρατηγικές βελτιστοποίησης που εξασφαλίζουν ομαλή διαχείριση των δεδομένων. Τα πειραματικά αποτελέσματα επιβεβαιώνουν την αποτελεσματικότητα του chatbot στην παροχή εξειδικευμένων απαντήσεων, αναδεικνύοντάς το ως μια ευέλικτη και αξιόπιστη λύση για ακαδημαϊκές ανάγκες.

Table of Contents

1. Introduction	5
1.1 Previous Research on Specialized Chatbots	5
2. Literature Review	7
2.1 Evolution of Chatbots: From Rule-Based to Neural Architectures	7
2.2 Retrieval-Augmented Generation (RAG) Methods and Principles	9
2.3 Token Management in GPT Models: Efficiency and Constraints	11
2.4 Context Clustering Techniques for Enhanced Retrieval	12
2.5 Hybrid Model Integration: Considerations for Domain-Specific Applications	14
3. Rag-Based Architecture and System implementation.....	17
3.1 Data Ingestion and Embedding Generation	17
3.1.1 Document Types (PDF, DOCX, CSV, XLSX, TXT) and Preprocessing Steps	18
3.1.2 Chunking Text and Generating Semantic Embeddings (SentenceTransformers) ..	20
3.1.3 Storing Embeddings and Metadata in MongoDB	22
3.2 Retrieval and Clustering Module	24
3.2.1 Vector Similarity Search for Domain-Specific Content	25
3.2.2 K-Means Clustering for Context Refinement.....	25
3.3 Generative Component for Specialized Chatbot Generation	28
3.3.1 Integration with OpenAI GPT API.....	29
3.3.2 Balancing Fluency, Accuracy, and Domain Relevance.....	29
3.3.3 Prompt Construction and Token Efficiency	30
3.3.4 Handling Ambiguity and Uncertainty	31
3.3.5 Continuous Improvement and Model Selection.....	31
3.4 Token Management and Prompt Construction.....	31
3.4.1 Importance of Token Constraints and Costs	32
3.4.2 Selecting Relevant Chunks	32
3.4.3 Hierarchical and Layered Prompt Designs	32
3.4.4 Adaptive Truncation Techniques	33
3.4.5 Maintaining Coherence and Logical Flow	33
3.4.6 Balancing Domain Specificity and General Guidance.....	33
3.4.7 Iterative Refinement and Evaluation.....	34
3.4.8 Future Directions in Token Management	34
3.5 Backend Implementation (Flask API) and Services Integration	34
3.5.1 Flask as the Integration Hub.....	34
3.5.2 Core Responsibilities of the Backend.....	35
3.5.3 Database and Embedding Service Integration.....	36
3.5.4 External Services and API Requests	36
3.5.5 Ensuring Performance, Scalability, and Reliability	36

3.5.6 Extensibility and Maintainability	37
3.6 Frontend Implementation (React Interface).....	37
3.6.1 Overview of the React-Based Architecture	37
3.6.2 Integration with Third-Party UI Libraries	37
3.6.3 Core Functionalities Reflected in the Code	38
3.6.4 Ensuring a Responsive and Intuitive UI	38
3.6.5 Accessibility and Device Compatibility	39
3.6.6 Scalability and Future Enhancements	40
3.6.7 Testing and Quality Assurance.....	40
4. Experiments	40
4.1 Domain – Specific Queries	40
4.2 Analysis of Results	49
5. Conclusion and Future Work	50
5.1 Summary of Contributions	50
5.2 Limitations and Potential Improvements	50
5.3 Future Directions for RAG-Based Specialized Chatbot Systems.....	51
6. References	51

1. Introduction

1.1 Previous Research on Specialized Chatbots

The landscape of chatbot technologies has undergone a profound metamorphosis over the past several decades, evolving from narrowly constrained rule-based systems to richly expressive, data-driven, and contextually aware conversational agents. Early chatbot exemplars, such as ELIZA (Weizenbaum, 1966) and PARRY (Colby, 1975), demonstrated the feasibility of machine-mediated dialogue at a time when computational resources and linguistic theories were comparatively nascent. These pioneering systems relied heavily on pattern matching, fixed heuristic rules, and scripted response templates, enabling rudimentary interactions that, while novel, remained limited in their depth, adaptability, and capacity to handle complex or domain-specific user queries. Although these early prototypes provided a seminal proof of concept, their inability to generalize beyond their programmed constraints underscored the need for more robust, flexible, and semantically aware models.

Subsequent generations of chatbots emerged against a backdrop of advancing computational linguistics and statistical machine learning methodologies. By integrating techniques such as Markov models, hidden Markov models, and later, conditional random fields, researchers introduced a measure of probabilistic reasoning into dialogue management and intent recognition. This statistical turn facilitated more nuanced interpretation of user inputs and partial separation from rigid rule sets. Nevertheless, these probabilistic frameworks still struggled with long-range semantic dependencies and relied on carefully engineered features, hindering their scalability and domain transferability. Their performance often plateaued when confronted with specialized content or evolving knowledge bases that demanded more contextual understanding and dynamic adaptation.

The introduction of neural network-based approaches, particularly recurrent neural networks (RNNs) and long short-term memory (LSTM) architectures (Sutskever et al., 2014), marked a turning point in the evolution of conversational agents. These early neural dialogue systems enabled end-to-end learning, reducing dependency on handcrafted features and manual tuning. Although they exhibited greater flexibility and improved fluency, they remained limited by difficulties in capturing extensive contextual nuances and handling the inherent ambiguity of natural language. The arrival of transformer-based architectures (Vaswani et al., 2017) and the subsequent proliferation of large language models (LLMs), such as BERT (Devlin et al., 2019) and GPT-3 (Brown et al., 2020), further revolutionized the field. These powerful models, trained on massive corpora, demonstrated remarkable linguistic fluency, general-purpose reasoning capabilities, and the ability to generate contextually coherent, human-like responses. They quickly established new standards for quality and versatility in open-domain dialogues.

Despite these notable strides, the deployment of chatbots in specialized domains has persistently posed significant challenges. Generic models, while excellent at open-domain tasks, often lack the domain-specific expertise, factual accuracy, and contextual grounding required in specialized environments. Whether assisting clinicians in interpreting medical guidelines, advising legal professionals on precedent-based rulings, guiding faculty and students through academic policies, or assisting industry specialists with intricate technical support, specialized chatbots must navigate complex, ever-changing knowledge ecosystems (Lee et al., 2021; Surden, 2019; Winkler & Söllner, 2018; Jain et al., 2018). Achieving high accuracy and domain relevance in these settings has traditionally necessitated fine-tuning pre-trained LLMs on targeted corpora (Howard & Ruder, 2018; Sun et al., 2019), a process that can be both computationally expensive and time-consuming. Moreover, reliance on static training data risks knowledge obsolescence, as domain information continuously evolves with updated guidelines, policies, and best practices.

In response to these constraints, research efforts have increasingly focused on Retrieval-Augmented Generation (RAG) architectures—an emerging class of models that integrate external retrieval mechanisms directly into the generative pipeline (Lewis et al., 2020). Rather than depending solely on internal model parameters for knowledge representation, RAG-based systems dynamically query external repositories, databases, or document collections at inference time. This design significantly reduces reliance on static training corpora, enabling chatbots to incorporate newly available information, respond to shifting institutional or industry standards, and maintain factual accuracy without the need for repetitive re-training. Early applications of these retrieval-augmented strategies gained traction in open-domain question-answering tasks (Chen et al., 2017; Karpukhin et al., 2020), where the ability to ground responses in authoritative sources proved essential to improving factual correctness and user trust. As these ideas matured, they naturally extended to specialized domains, thereby reinforcing the value of retrieval-based approaches for domain-specific chatbot deployments.

Parallel refinements to RAG systems have tackled other essential components of high-quality specialized chatbots. One line of inquiry focuses on employing semantic embeddings and advanced clustering techniques to improve the precision of document retrieval (Reimers & Gurevych, 2019; Izacard & Grave, 2021). By grouping related text segments into semantically coherent clusters and selecting contextually aligned passages, chatbots can supply generative models with richer, denser context. This approach not only enhances accuracy and relevance but also helps mitigate token overhead within the model's limited context window.

In tandem with improved retrieval and clustering strategies, researchers have turned their attention to efficient prompt engineering, token management, and computational resource optimization to maintain a delicate balance between performance and latency (Liu et al., 2021). These techniques involve carefully selecting and arranging context segments, truncating unnecessary information, and leveraging hierarchical prompt designs or multi-stage inference pipelines. The resulting prompts are more token-efficient and cost-effective, ensuring that high-quality outputs do not come at the expense of disproportionate computational costs or sluggish response times.

Collectively, these lines of inquiry have advanced the state of specialized chatbot research by demonstrating that domain-relevant accuracy, factual robustness, and contextual nuance are achievable within complex, evolving information landscapes. By integrating semantic retrieval layers, deploying sophisticated clustering algorithms for thematic refinement, and employing strategic token management, today's specialized chatbots can deliver authoritative, context-aware, and user-centric responses. The academic, professional, and industrial implications of these breakthroughs are profound, as domain-specialized chatbots become an increasingly viable solution for supporting expert-level interactions, providing reliable guidance, and streamlining access to relevant knowledge.

This thesis builds upon and extends these insights. By focusing on Retrieval-Augmented Generation architectures and their associated retrieval, clustering, and token optimization techniques, it aims to refine both the theoretical foundations and practical implementations of specialized chatbots. In doing so, this work contributes to the ongoing transformation of conversational AI into a cornerstone technology for academic institutions, enterprises, and professional communities that demand adaptive, trustworthy, and domain-specific dialogue agents capable of meeting the evolving needs of highly specialized environments.

2. Literature Review

2.1 Evolution of Chatbots: From Rule-Based to Neural Architectures

The evolution of chatbot technologies has followed a trajectory marked by continual refinement, conceptual innovation, and paradigm shifts in both methodology and technology. Early conversational agents, typified by canonical systems such as ELIZA (Weizenbaum, 1966) and PARRY (Colby, 1975), laid the foundational groundwork for what would eventually become a thriving field at the intersection of artificial intelligence, natural language processing, and human-computer interaction. These pioneering models, developed at a time when computational resources and linguistic theory were far more limited, relied almost exclusively on deterministic, rule-based approaches. In practice, they implemented sets of handcrafted patterns, simple keyword triggers, and scripted responses. Such mechanisms enabled rudimentary forms of simulated dialogue that emulated specific conversational roles or disorders, as in the case of PARRY's modeling of paranoid behavior. Yet, these systems were inherently constrained. They lacked any genuine understanding of language semantics or user intent and were highly sensitive to unexpected inputs. Consequently, early chatbots often produced repetitive, shallow, and easily derailed interactions, providing a first glimpse of automated dialogue but offering limited flexibility, adaptability, and contextual awareness (Shum et al., 2018).

Subsequent advancements in the latter decades of the 20th century and the early 2000s sought to overcome the brittleness and rigidity of these rule-based architectures. Statistical methodologies and emerging machine learning techniques began to reshape the landscape of conversational agents. Rather than depending solely on pre-defined rules, researchers experimented with data-driven frameworks that leveraged annotated corpora, probabilistic models, and statistical methods to capture linguistic structures and dialogic patterns more dynamically. Approaches incorporating Markov models, hidden Markov models (HMMs), and later conditional random fields (CRFs) were developed to enhance intent recognition, dialogue management, and response selection (Jurafsky & Martin, 2021). These probabilistic frameworks introduced a degree of flexibility and allowed for more nuanced interpretation of user inputs. Nevertheless, they still encountered significant challenges. For instance, while such models could handle some degree of variation in user queries, they remained limited in their capacity to understand long-range dependencies, track evolving conversation states beyond a few turns, or infer implied context not explicitly stated. Their reliance on handcrafted features and domain-specific heuristics further restricted scalability, generality, and adaptability to new domains without extensive manual engineering.

A substantial leap forward in the evolution of chatbot architectures emerged with the introduction of neural network-based models, which began to gain prominence in the mid-2010s. Early neural conversational agents often leveraged recurrent neural networks (RNNs), particularly long short-term memory (LSTM) units and gated recurrent units (GRUs), to better capture temporal dependencies and maintain a representation of conversational context over multiple turns (Sutskever et al., 2014; Cho et al., 2014). This end-to-end learning paradigm freed researchers from the burden of manual feature engineering, allowing chatbots to learn conversational patterns directly from large volumes of raw text data. The neural approach enabled improvements in fluency, coherence, and adaptability, giving rise to systems that could, to some extent, handle a wider range of user inputs, display rudimentary context retention, and produce responses that felt more natural and less templated than their rule-based predecessors. Despite these gains, early neural dialogue systems still faced limitations, including difficulties in producing genuinely informative answers, maintaining consistency over extended interactions, and handling rare or out-of-distribution inputs. The complexity and ambiguity inherent in natural language, coupled with the scarcity of large, high-quality dialogue

corpora for specialized domains, meant that these neural approaches often plateaued in performance for intricate, domain-specific tasks.

The advent of transformer-based architectures (Vaswani et al., 2017) revolutionized the field of natural language processing and, by extension, chatbot development. Unlike RNNs, transformers rely on multi-headed attention mechanisms to model dependencies between words at arbitrary distances, thereby excelling in capturing long-range contexts, nuanced semantic relations, and subtle pragmatic cues. These architectures facilitated massive parallelization, enabling training on unprecedentedly large and diverse corpora. As a result, transformer-based language models, notably BERT (Devlin et al., 2019), GPT-2 (Radford et al., 2019), GPT-3 (Brown et al., 2020), and subsequent variants, have dramatically expanded the capabilities of conversational agents. Such models exhibit sophisticated linguistic understanding, can generate responses with human-like fluency, and appear to encode a broad range of world knowledge acquired through large-scale pre-training. This leap has established new benchmarks for coherence, informativeness, and linguistic versatility in chatbot systems.

Yet, as transformative as these models have been, significant challenges persist. While large-scale language models achieve remarkable performance in open-domain dialogue and general knowledge tasks, they often falter when faced with specialized domains that require precise, factual accuracy and alignment with domain-specific ontologies or policies. For example, a university-oriented chatbot may need to reference official regulations, departmental guidelines, administrative procedures, or dynamically updated course lists—resources not easily encapsulated in model parameters trained predominantly on general web data. In these cases, reliance on the model's internal weights for factual correctness leads to issues such as hallucinations, outdated information, and a mismatch between user needs and provided answers. This discrepancy highlights the ongoing need for retrieval-augmented strategies, domain adaptation approaches, and token optimization techniques that efficiently integrate external, authoritative data sources into the generative process (Lewis et al., 2020; Izacard & Grave, 2021). Research in this direction aims to augment large language models with retrieval modules that fetch relevant, domain-specific documents at inference time, ensuring that the resulting answers remain grounded in verified knowledge rather than speculative reasoning.

As this historical perspective indicates, chatbot evolution has progressed from static, rule-bound constructs with severely limited conversational ability to dynamic, data-driven, and contextually enriched architectures powered by state-of-the-art language models and retrieval augmentation. Each stage of development—rule-based scripting, statistical modeling, neural sequence modeling, and transformer-based generative models—has contributed key methodologies, conceptual frameworks, and technical tools. These collective advancements have laid a robust foundation for today's sophisticated conversational agents, which are increasingly capable of engaging in rich, multi-turn dialogues, adapting to new domains with relatively less manual overhead, and delivering nuanced, context-aware responses.

The trajectory from simplistic pattern matching to complex, domain-aware generative reasoning sets the stage for the subsequent sections of this thesis. The focus now shifts toward understanding how Retrieval-Augmented Generation (RAG) and related innovations—such as context clustering, token management, and hybrid inference strategies—can be leveraged to produce specialized, domain-focused chatbots. These next-generation systems aim not only for human-like fluency but also for factual accuracy, trustworthiness, adaptability, and the capacity to integrate seamlessly with evolving knowledge ecosystems. In doing so, they represent the culmination of decades of progress and serve as a blueprint for future research and development in deploying conversational AI for specialized, mission-critical applications.

2.2 Retrieval-Augmented Generation (RAG) Methods and Principles

Retrieval-Augmented Generation (RAG) represents a substantive shift in the design principles and operational paradigms of conversational AI systems, enabling them to bridge the gap between the static internal knowledge captured by pre-trained language models and the dynamic, ever-evolving world of external information sources. Unlike conventional generative approaches that rely predominantly, if not exclusively, on massive neural models and their parameterized memories of linguistic and factual data, RAG architectures incorporate a retrieval mechanism that actively pulls pertinent knowledge from external repositories. This strategic integration addresses a fundamental shortcoming of large language models (LLMs): while they exhibit remarkable linguistic fluency, creativity, and broad reasoning capabilities, they struggle to consistently deliver accurate, time-sensitive, and domain-specific information without extensive and costly retraining or fine-tuning.

A key motivation behind the RAG approach arises from the recognition that purely parametric models, even those with billions of parameters, have limited and static knowledge horizons. Such models are trained on large-scale corpora that represent a snapshot of human knowledge at the time of data collection. As a result, facts can become outdated, and domain-specific nuances often remain underrepresented unless explicitly included and emphasized in the training data. Fine-tuning on specialized corpora—once considered a straightforward solution to domain adaptation—proves cumbersome. It demands substantial computational resources, reintroduces engineering overhead, and risks knowledge staleness as soon as the target domain evolves. This challenge is amplified in fields such as university administration, legal compliance, medical information, or industrial standards, where authoritative documents and policies change regularly.

RAG-based architectures directly tackle these issues. Instead of depending solely on model-internal parameters to generate contextually relevant answers, they integrate an external retrieval process at inference time (Lewis et al., 2020). In practice, this means that when a user issues a query—be it a question about a university’s admission deadlines or a request for a specific regulatory guideline—the system first encodes the query into a high-dimensional vector representation. Such vector embeddings are produced by advanced sentence-level encoders, often leveraging transformer-based models fine-tuned for semantic similarity tasks, such as SentenceTransformers (Reimers & Gurevych, 2019) or Dense Passage Retrieval (DPR) encoders (Karpukhin et al., 2020). These embeddings capture subtle semantic relationships and user intent, enabling the system to sift through large corpora of documents, text segments, or knowledge graph entries to identify the most relevant pieces of content.

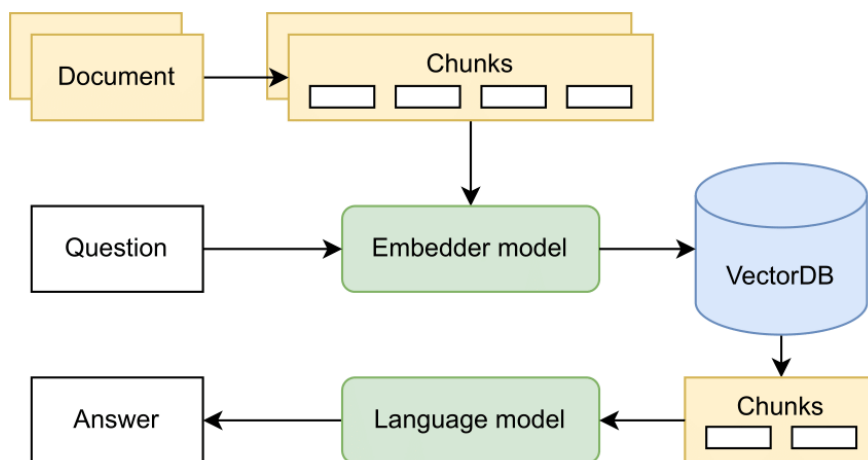


Figure 1 Retrieval-Augmented-Generation-Architecture

Once the top-ranked documents or passages are retrieved, the generative language model receives this curated context alongside the user's query. Crucially, this retrieval step occurs at runtime, allowing the system to adapt to newly added or updated documents without necessitating model retraining. By conditioning on authoritative, domain-specific content, the language model's output is "grounded" in evidence rather than drawn from potentially outdated or incomplete internal parameters. This grounded approach reduces hallucination, where models fabricate facts, and fosters transparency and trustworthiness. The chain-of-thought behind answers can be inspected, as the retrieved documents are accessible and can be updated. If, for example, a university changes its admission policies, administrators need only revise the underlying documents; the chatbot, through its retrieval step, will incorporate these changes immediately into its answers.

Another cornerstone principle of RAG is modularity. By separating retrieval from generation, researchers and practitioners gain the freedom to optimize each component independently. The retrieval layer can be improved by experimenting with different embedding models, re-ranking strategies, or indexing structures. Meanwhile, the generative layer can progress with the development of more powerful language models or novel prompt engineering techniques. This modular design transforms the system into a flexible ecosystem of interacting parts rather than a monolithic entity tied to a single model's capabilities.

The inherent adaptability of RAG systems also leads to opportunities for token management and computational efficiency. Since large language models have finite context windows and associated inference costs, retrieving only the most relevant subsets of the data ensures that the context fed into the model remains dense and meaningful. Techniques such as hierarchical retrieval, context clustering (Izacard & Grave, 2021), and semantic filtering can prune extraneous content, mitigating the risk of exceeding token limits. Token optimization thus emerges as a complementary practice to RAG, ensuring that each token expended in the prompt is justified by its contribution to the final answer's accuracy and relevance (Liu et al., 2021). By doing so, organizations deploying these systems can maintain manageable costs, achieve faster response times, and improve user satisfaction.

Empirical evidence supports the efficacy of retrieval augmentation. Studies across multiple domains underscore how RAG-based solutions outperform purely generative approaches in factual correctness, reliability, and domain specialization. For instance, in open-domain question answering (Chen et al., 2017), combining retrieval and generation leads to state-of-the-art performance, as the model no longer depends solely on pre-training knowledge to answer queries about arbitrary topics. In enterprise or technical support settings (Jain et al., 2018), RAG methodologies allow chatbots to handle product documentation, troubleshooting guides, and user manuals that evolve over time. Instead of re-training the entire model after each documentation update, the system can pull from the latest materials. Similarly, academic or professional information services (Winkler & Söllner, 2018) benefit when chatbots dynamically reference official handbooks, course catalogs, or policy repositories.

This retrieval-driven paradigm also opens the door to more interpretable and audit-friendly AI. Since the ultimate response is generated from explicitly retrieved content, users and maintainers can trace back the model's answers to their original sources, validating their integrity or identifying errors. Such interpretability aligns well with emerging regulations and ethical guidelines that encourage transparency in AI-powered decision-making systems. For institutions like universities, where credibility and trust are paramount, having a chatbot whose answers can be verified or updated by adjusting the source documents builds confidence and ensures that the system remains aligned with institutional standards.

Furthermore, RAG-based architectures are not limited to text-based information retrieval. Ongoing research explores how these principles could extend to multimodal domains,

allowing models to query image databases, structured spreadsheets, or specialized knowledge graphs. In academic contexts, this might mean pulling a relevant figure from an online course platform or referencing a dataset that supports a particular research claim. The underlying principle—retrieval at inference time—remains consistent, ensuring that generation is always tethered to solid, external grounding.

As the field progresses, we may see even more granular control over retrieval, including dynamic reranking of results based on conversational context, user role, or adaptive thresholds that adjust the number of retrieved documents according to query complexity. Data curation practices will become increasingly integral, as maintaining a high-quality knowledge base ensures that retrieval-augmented answers remain authoritative. Moreover, integration with techniques like active learning or feedback loops can continuously refine the retrieval model, making it increasingly adept at identifying what is relevant and ignoring what is not.

In conclusion, Retrieval-Augmented Generation (RAG) establishes a new design paradigm that transcends the limitations of purely parameter-centric language models. By embedding retrieval steps into the generative process, RAG fosters adaptability, ensures long-term maintainability of domain-specific knowledge, and encourages more efficient computational strategies. The synergy between retrieval and generation, grounded in empirical validations and a growing body of research, defines RAG as a versatile, future-ready approach to building sophisticated chatbots and conversational systems capable of delivering not just linguistic grace, but also factual correctness, topical relevance, and continuous adaptability in an ever-changing information landscape.

2.3 Token Management in GPT Models: Efficiency and Constraints

The capabilities of large language models (LLMs) such as GPT-3.5 and GPT-4 have improved the fluency and coherence of machine-generated text; however, these models operate within practical constraints imposed by their maximum context windows. Tokens—subdivisions of textual input and output—directly influence both the cost and feasibility of engaging with such models. Exceeding the model's context window, which typically ranges from a few thousand to tens of thousands of tokens, can result in truncated input, incomplete reasoning, or outright inability to process the request. Consequently, effective token management has emerged as a crucial consideration in the literature on dialogue systems and RAG-based architectures.

Token management encompasses strategies for optimizing which information is retained, truncated, or omitted in the prompt construction phase to ensure that essential context is conveyed to the model within its operational limits. Early work on token management often involved naive truncation policies, simply discarding trailing content once the token budget was reached. However, such approaches risk removing critical background information and destabilizing the model's response quality (Liu et al., 2021).

To address these limitations, researchers have developed more sophisticated strategies centered on semantic relevance and information density. One family of approaches leverages embedding-based similarity scores to rank candidate context segments and include only those passages most germane to the user's query. By employing semantic embeddings derived from models such as Sentence Transformers (Reimers & Gurevych, 2019), systems can prioritize thematically aligned documents that are likely to enhance the model's grounding and factual accuracy. This ensures that irrelevant or repetitive segments are excluded, making space for more pertinent text within the model's token constraints.

Another dimension of token management pertains to the structuring of prompts. Hierarchical or layered prompt designs can compress multiple related documents into summarized forms before integrating them into the primary query prompt (Yang et al., 2021). Such hierarchical strategies allow the system to present the model with a distilled version of

extensive background material without exceeding token limits. Additionally, dynamic thresholding techniques can be applied to adaptively determine how much context to include based on the complexity of the user's query and current computational constraints, striking a balance between richness of information and prompt compactness.

These refined token management practices carry significant practical implications, particularly in resource-constrained environments where API usage costs and latency must be carefully managed. Optimizing token usage reduces unnecessary computation and associated expenses, while simultaneously enhancing user experience by providing focused, contextually informed responses. The capacity to adapt the prompt length and content on-the-fly also contributes to system robustness and scalability, attributes essential for large-scale deployments in academic and enterprise settings.

Token management is not merely a technical detail but a central design choice in RAG-based chatbot systems. By thoughtfully curating context, leveraging semantic embeddings for content selection, and employing hierarchical prompt construction, researchers and practitioners can ensure that LLMs operate efficiently within their token constraints. This optimization ultimately enhances response accuracy, reduces computational overhead, and enables specialized chatbots to reliably deliver timely, domain-specific information even when confronted with extensive and evolving knowledge repositories.

2.4 Context Clustering Techniques for Enhanced Retrieval

A central challenge in Retrieval-Augmented Generation (RAG) architectures lies not merely in identifying documents pertinent to a user's query, but in doing so in a manner that ensures the collected context segments complement one another, collectively offering a rich, coherent, and contextually comprehensive basis for the generative model's response. While advanced dense vector retrieval methods, such as those leveraging bi-encoder architectures or dual-tower encoders (Karpukhin et al., 2020; Reimers & Gurevych, 2019), have demonstrated substantial gains in pinpointing semantically relevant text units, these methods alone may still return redundant, near-duplicate passages or inadvertently mix thematically disjoint material. Such issues become particularly problematic as the knowledge corpus grows in size, complexity, and thematic breadth.

In this regard, context clustering techniques have emerged as a powerful refinement strategy within the RAG pipeline. Clustering involves grouping semantically similar text chunks—whether sentences, paragraphs, or short document segments—into cohesive sets that represent distinct topical dimensions or subthemes within the corpus. By imposing structure on an otherwise amorphous body of retrieved content, clustering methods help ensure that the generative model receives context that is not only highly relevant but also free from unnecessary repetition. For instance, rather than presenting the model with five slightly varying passages that all describe the same administrative procedure, a clustering-based approach might select one high-quality representative passage from that cluster. This step not only enhances the semantic density of the context but also conserves valuable token space within the language model's input window, a critical resource given token limit constraints and associated computational costs.

A variety of clustering algorithms can be applied, each with its own strengths, weaknesses, and suitability for particular data distributions. K-means clustering, one of the most well-known and computationally efficient algorithms, partitions embedding vectors into a predefined number of clusters (Izacard & Grave, 2021). Hierarchical clustering, by contrast, constructs a tree-like structure of nested clusters, enabling more flexible exploration of thematic granularity: one can drill down into finer subtopics as needed. Density-based methods like DBSCAN or HDBSCAN (Campello et al., 2013) allow for the discovery of clusters of arbitrary shape and the identification of outliers, making them useful when dealing with highly

heterogeneous corpora that do not conform to spherical cluster assumptions. Choosing the right clustering algorithm depends on the nature of the domain, the size of the corpus, and the intended retrieval strategies.

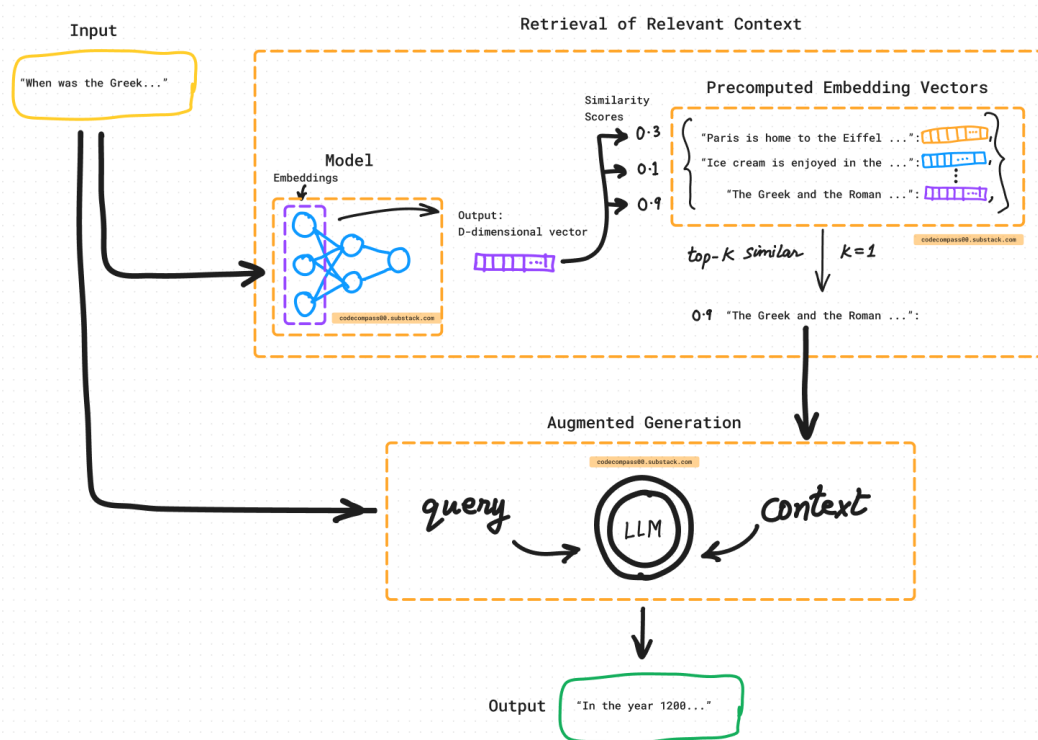


Figure 2. Retrieval-Augmented-Generation-Architecture with clustering

Central to effective clustering is the quality of the underlying embeddings. Embeddings derived from transformer-based encoders, especially those fine-tuned for semantic similarity tasks or domain-specific objectives, can capture subtle lexical and syntactic nuances, as well as thematic and conceptual associations (Gao et al., 2021). High-quality embeddings ensure that chunks discussing related topics—such as specific academic programs, departmental policies, or research methodologies—gravitate toward one another in the vector space. As a result, the clustering process becomes more meaningful and reliable, carving the corpus into semantically coherent regions rather than arbitrary groupings. Without robust embeddings, clusters may become noisy or reflect superficial lexical patterns rather than genuine thematic consistency, undermining the primary benefit of the approach.

Once clusters are formed, retrieval processes can adopt a multi-stage approach. Instead of searching the entire corpus for relevant chunks, the system may first identify which clusters align most closely with the user's query embedding. The query-to-cluster relevance step filters out large swaths of irrelevant data before any fine-grained selection of individual documents occurs. After narrowing down the search to a handful of highly relevant clusters, the system can then retrieve the top-ranked chunks within those clusters. This hierarchical narrowing of scope improves both the speed and the accuracy of retrieval. Rather than sifting through thousands of candidate passages, the system primarily considers the subset of the corpus that is thematically aligned with the user's request, thereby reducing computational overhead and improving latency.

Beyond retrieval optimization, clustering also provides operational and maintenance advantages. Clusters serve as a form of topical indexing over the knowledge base, offering

insights into the domain's structure and helping curators understand how documents group together. This top-level view can aid in content curation: when new documents are added or existing ones become obsolete, administrators can re-run the clustering process incrementally or adopt streaming clustering methods that integrate fresh data into existing clusters without a full re-indexing of the corpus. Such adaptability is crucial in domains subject to continual updates, where policies, regulations, or course offerings change periodically. By updating or refining clusters, the system maintains thematic integrity and ensures that retrieval remains sharply focused on the most current and authoritative material.

From a design perspective, clustering interfaces smoothly with the broader RAG ecosystem. Token management techniques, discussed in Section 2.3, complement clustering by helping ensure that the selected clusters' representative passages fit neatly into the language model's input constraints. By feeding the model fewer but higher-quality chunks, the system fosters more coherent and factual answers. Generative models, in turn, benefit from the thematic consistency that clusters provide, as they can draw on semantically related passages that reinforce each other's context rather than distract with conflicting or redundant information.

In practical terms, consider a scenario within a university domain: an academic chatbot must answer questions about a range of topics, from admission requirements and scholarship opportunities to research guidelines and campus events. Without clustering, the retrieval layer might return multiple passages covering the same undergraduate admission policy. While these passages are relevant, their repetitive nature wastes token space and may force the generative model to choose among them arbitrarily, risking inconsistency or confusion. By applying clustering, the system identifies a set of semantically coherent chunks—one cluster might focus on admissions, another on graduate fellowships, a third on research ethics—and selects a single, high-quality representative for each theme. The end result is a curated prompt that provides the model with diverse yet complementary contextual snippets, fostering richer, more authoritative responses.

Clustering can also reveal gaps or imbalances in the corpus. If certain queries consistently direct retrieval into "sparse" clusters—those containing fewer or lower-quality documents—this may signal a need to enrich the knowledge base. Administrators might respond by adding relevant documents or reorganizing existing materials. Thus, clustering not only refines retrieval but also guides strategic corpus management and continuous improvement of the system's domain coverage.

In conclusion, context clustering techniques serve as a pivotal refinement layer in RAG-based architectures, bridging the gap between raw semantic retrieval and finely tuned, domain-specialized knowledge delivery. By organizing corpora into semantically coherent groups, clustering promotes efficiency, context relevance, token optimization, and ultimately user satisfaction. As domains grow in complexity and corpora expand, clustering stands as a critical step that ensures the RAG pipeline remains both effective and adaptable, enabling next-generation chatbots to deliver genuinely helpful, contextually aware answers that meet the intricate demands of specialized informational ecosystems.

2.5 Hybrid Model Integration: Considerations for Domain-Specific Applications

As the field of conversational AI advances, there is a growing recognition that no single model or infrastructure configuration can perfectly meet the multifaceted demands of specialized, dynamic, and resource-sensitive environments. While large language models (LLMs) have made remarkable strides in general fluency and broad-coverage reasoning, relying exclusively on a monolithic system can be suboptimal, especially where accuracy, domain alignment,

trustworthiness, and operational cost management are paramount. Hybrid model integration has thus emerged as a strategic response, blending multiple model types, retrieval layers, and infrastructural components into a unified, context-aware conversational agent. The result is a more flexible, tunable architecture that can adapt to evolving requirements, user profiles, and knowledge ecosystems.

One central consideration in hybrid architectures concerns the balance between external APIs and local, in-house computational resources. Publicly available, cloud-based LLM services—exemplified by systems such as the OpenAI GPT API—provide cutting-edge linguistic abilities and rapid deployment capabilities without the overhead of managing large-scale model training and hosting infrastructures (Bommasani et al., 2021). For organizations seeking immediate sophistication, this path offers a tempting shortcut. However, reliance on external APIs can introduce several trade-offs. Latency issues may arise if queries must travel across geographic distances, potentially affecting user experience. Costs can escalate with increasing query volumes, and ongoing subscription fees or pay-per-query pricing models may strain institutional budgets over time. Furthermore, data privacy and regulatory compliance are significant considerations. Universities, healthcare institutions, government agencies, and other regulated organizations often maintain strict policies regarding data sovereignty and confidentiality. In such cases, routing sensitive queries through external services might conflict with compliance frameworks or raise ethical concerns.

By contrast, hosting smaller, more focused models locally grants organizations full control over their data, latency, and scaling strategies. Locally deployed models can be tailored to the specific vocabulary, norms, and content of the domain, ensuring that sensitive institutional documents never leave the organization's secure environment. Yet, these locally hosted models frequently lag behind their cloud-based counterparts in linguistic sophistication and may demand ongoing maintenance, hardware investment, and retraining efforts. The hybrid approach mitigates these dilemmas by dynamically routing queries to the most suitable inference endpoint. A low-stakes query—such as asking for office hours or course listing details—could be handled by a smaller, cost-effective local model. Meanwhile, complex academic inquiries requiring nuanced reasoning and domain adaptation might be directed to a remote state-of-the-art API. This orchestration ensures that the system continuously aligns resource expenditure with query complexity, thereby balancing performance with cost-effectiveness.

Another dimension of hybrid integration involves managing diverse retrieval pipelines and knowledge sources. Instead of relying on a single repository, a hybrid system can incorporate multiple, thematically distinct data silos. For instance, consider the complex environment of a university. Some queries might relate to stable and static information, such as official policy documents or historical syllabi. Others might concern rapidly changing data—like announcements for upcoming seminars, new research publications, or updated administrative procedures. A hybrid retrieval strategy could direct stable, regulation-focused queries to a well-curated, versioned database containing authoritative policy documents, while queries seeking current information about departmental events or recently approved courses could be routed to a dynamically refreshed corpus of announcements or RSS feeds. By intelligently merging these parallel retrieval streams, the system ensures that the final prompt to the generative model presents a coherent blend of static, high-authority texts and timely, contextually relevant updates. The system can thus deliver responses that accurately reflect the current state of the domain, a critical requirement in fast-evolving academic or professional fields (Chen et al., 2017; Karpukhin et al., 2020).

Hybrid integration also supports improved model interpretability and reliability through multi-model ensembles and verification layers. For example, the system may first generate a response using a high-capacity LLM and then pass this output to a secondary, more

conservative model or a specialized re-ranker for verification. If the second model detects potential factual inaccuracies or domain mismatches, it may trigger a fallback strategy—retrieving additional context or substituting a simpler local model’s answer. By establishing multiple “checkpoints” in the answer pipeline, hybrid architectures reduce the risk of error propagation and ensure that each final response is not only contextually aligned but also rigorously vetted and cross-validated. In academic settings, where misinformation or outdated guidance can have serious consequences, such safeguards help maintain user trust and system credibility (Liu et al., 2021).

Cost and computational efficiency remain central considerations. Hybrid systems allow administrators to finely calibrate resource allocation to match query difficulty, user volume, and seasonal fluctuations. For instance, during exam registration periods, a university chatbot might experience a surge in queries related to deadlines and prerequisites—topics that might be adequately handled by a locally hosted model using previously retrieved and frequently accessed documents. Such an approach keeps latency low and costs stable. Conversely, when occasional complex queries arise—e.g., in-depth inquiries about interdisciplinary research opportunities—a more expensive but more knowledgeable API-driven LLM can be invoked selectively. Over time, institutions can develop heuristic policies or even machine-learning-driven heuristics that continuously optimize these routing decisions based on historical usage patterns, evolving corpus structure, and observed user satisfaction metrics.

Beyond cost and efficiency, hybrid integration may also encompass personalization and adaptation to user roles. Consider a university scenario once more: faculty, administrative staff, undergraduate students, graduate researchers, and external visitors may each pose different types of queries. By incorporating user identification or authentication layers, the chatbot can route queries differently depending on the user profile. A faculty member, for instance, might have permission to access certain restricted databases, enabling the retrieval module to pull from specialized sources. A hybrid system could use a local model specialized in departmental governance documents for faculty queries while relying on a public API for more general, open-domain knowledge requested by students. Such role-based customization further refines the adaptability and utility of the system.

As research in Retrieval-Augmented Generation (RAG), token management, and clustering matures, hybrid integration strategies will likely evolve to incorporate more sophisticated orchestration mechanisms. For example, future systems may leverage reinforcement learning to continuously refine routing policies or incorporate meta-models that predict which combination of models and retrieval streams will yield the best outcome for each query. Advanced caching strategies, where frequently requested information is pre-computed or stored locally for rapid retrieval, can further enhance responsiveness and reduce overhead. Additionally, hybrid integration frameworks may incorporate external monitoring and analytics tools, allowing system maintainers to track performance across various dimensions—latency, accuracy, cost, user satisfaction—and adjust the infrastructure accordingly.

In sum, hybrid model integration stands at the intersection of performance, reliability, adaptability, and transparency. By combining the strengths of external APIs with the control and stability of local infrastructures, and by orchestrating multiple retrieval pipelines, hybrid architectures offer a robust toolkit for addressing the nuanced challenges of specialized chatbot deployments. As conversational AI steadily grows more complex and domains continue to diversify, these hybrid strategies promise to become increasingly essential, empowering developers and stakeholders to deliver specialized, authoritative, and context-sensitive conversational experiences that seamlessly align with the evolving demands of academic and professional environments.

3. Rag-Based Architecture and System implementation

3.1 Data Ingestion and Embedding Generation

The foundation of any Retrieval-Augmented Generation (RAG)-based system rests on the quality, coherence, and accessibility of its underlying knowledge base. In the present implementation, data ingestion and embedding generation have been designed to ensure that the chatbot can effectively ground its responses in authoritative, domain-specific information. This process involves three primary stages: document preprocessing, text segmentation, and semantic embedding generation, culminating in the storage of these enriched data units within a scalable repository.

The initial step entails collecting and consolidating diverse document formats, including PDF, DOCX, CSV, XLSX, and TXT files. Each source undergoes a standardized preprocessing pipeline to extract clean, machine-readable text. PDF documents, for instance, are parsed using dedicated libraries (e.g., PyPDF2) to retrieve their textual content, while DOCX and tabular files (such as Excel or CSV) are similarly processed through corresponding parsers. The primary goal at this stage is to ensure that all textual sources are transformed into a consistent, uniform representation, minimizing noise introduced by varying data formats, extraneous characters, or layout artifacts.

Once the corpus is normalized, the textual data is segmented into manageable “chunks” of a predetermined size (e.g., approximately 500 words). This segmentation serves multiple purposes. Firstly, it enhances retrieval efficiency and semantic precision, as smaller, self-contained chunks can be more readily matched to user queries than larger, heterogeneous text blocks. Secondly, chunking facilitates the downstream embedding step, ensuring that embeddings represent focused, contextually coherent units of information rather than sprawling, topically mixed segments. This granularity is particularly critical in specialized domains, where users may pose highly targeted questions that align more closely with specific subsections of a document.

After segmentation, each text chunk is converted into a numerical vector representation via a transformer-based embedding model, such as SentenceTransformers (Reimers & Gurevych, 2019). These embeddings encode semantic relationships and contextual nuances, enabling similarity comparisons and relevance rankings that transcend simple keyword matching. By mapping natural language text into a high-dimensional vector space, the system can effectively measure the semantic distance between a user query and various document segments, identifying which chunks best address the informational need at hand.

In selecting and fine-tuning the embedding model, factors such as domain specificity, multilingual capabilities, and computational efficiency are considered. Although general-purpose embedding models often prove sufficient for many use cases, specialized domains may benefit from embeddings fine-tuned on domain-relevant corpora or tailored architectures that capture professional jargon, institutional policies, or technical terminology. Integrating a robust embedding model ensures that the RAG system can accurately retrieve and rank the most relevant chunks, guiding the generative component toward factually grounded responses.

Finally, the embedding vectors, along with their associated text, filenames, and metadata, are stored in a MongoDB database. MongoDB’s flexible schema and indexing capabilities facilitate efficient data management, update operations, and scalability. This architecture not only streamlines subsequent retrieval operations, but also simplifies maintenance tasks such as adding new documents, updating outdated information, or removing redundant chunks. By maintaining embeddings and their textual counterparts in a centralized, query-friendly environment, the system ensures that domain knowledge remains accessible and adaptable to evolving requirements.

3.1.1 Document Types (PDF, DOCX, CSV, XLSX, TXT) and Preprocessing Steps

A key challenge in designing a Retrieval-Augmented Generation (RAG)-based chatbot system lies in ensuring that domain-relevant information is accurately and efficiently extracted from a variety of source documents. The heterogeneity of formats commonly found in real-world knowledge repositories—ranging from portable document formats (PDF) and word processing files (DOCX) to tabular data (CSV, XLSX) and plain text (TXT)—necessitates a robust and methodical approach to preprocessing. In this section, we detail the motivations, tools, and methodologies employed to transform raw documents into clean, normalized text ready for embedding and subsequent retrieval operations.

The University of Piraeus, like many academic institutions, hosts an extensive and evolving body of documentation. These materials might include official regulations, administrative guidelines, course descriptions, research abstracts, policy statements, and event announcements. Academic stakeholders and students often consult such documents to answer queries related to admissions procedures, curriculum details, departmental policies, and scholarly activities. These texts are rarely uniform in format. Some are circulated as PDFs, others as DOCX files, and still others as spreadsheets. Over time, the corpus may expand to include data exported from internal databases as CSV or XLSX files, as well as legacy documents retained in plain text (TXT) form. Capturing knowledge from this multiplicity of formats is essential to developing a truly comprehensive and authoritative chatbot capable of managing a wide range of user inquiries.

File Type Considerations:

PDFs are widely used for final or official documentation due to their platform-independent, fixed-layout format. However, their internal structure does not always lend itself to straightforward text extraction. Visual formatting elements such as headers, footers, multi-column layouts, and embedded images can complicate parsing. Additionally, not all PDF files are text-based; some rely on scanned images or contain selective text layers. To address these complexities, specialized libraries (e.g., PyPDF2) are employed to decode PDF content. Where the document is text-based, the library extracts textual segments directly. In more challenging cases, Optical Character Recognition (OCR) techniques could be applied, though this may introduce noise and inaccuracies. For the purpose of this project, only PDF files with embedded text layers are considered to ensure a consistent level of extraction quality.

DOCX is a common format for editable text documents, including syllabi, research summaries, administrative memos, and meeting minutes. Unlike PDFs, DOCX files typically retain semantic and structural information through XML-based representations. Libraries such as `python-docx` facilitate the extraction of paragraph-level text with relative ease. While formatting elements (e.g., font styles, headings) are not directly preserved in the resulting text strings, their hierarchical structuring allows for a cleaner textual output. Thus, DOCX extraction generally yields well-structured and complete textual representations suitable for downstream segmentation and embedding.

Tabular data formats introduce unique considerations. CSV and XLSX files often contain highly structured information—such as course lists, faculty directories, or timetables—organized into rows and columns. Extracting meaningful textual content from these sources involves converting each row into a textual representation. By using `pandas` (a Python data analysis library), spreadsheets can be loaded into data frames and then converted to a human-readable string format. While this approach may flatten the inherent relational structure of the data, it retains essential textual content. Ensuring that column headers, cell values, and numerical data are combined into comprehensible segments is crucial. Careful preprocessing

can transform what initially appears as purely tabular data into a narrative-like format suitable for semantic embedding. For instance, a CSV listing course codes, titles, instructors, and schedules can be translated into textual segments that combine these attributes into a coherent sentence, improving retrieval accuracy when the chatbot handles queries such as “Which professors teach advanced data analytics?”

TXT files present the least complexity in terms of extraction. They do not require specialized parsing, as their content is already in a raw textual form. However, the absence of structural markers and formatting cues means that these documents may be less organized. TXT files might include free-form notes, manually typed reference lists, or legacy materials that have not been structured. Preprocessing these documents primarily involves trimming whitespace, removing extraneous characters (such as non-ASCII symbols if not required), and ensuring consistency in character encoding (e.g., UTF-8) to prevent corruption of non-Latin scripts or accented characters. Although plain text is simplest to handle, care must be taken to segment it effectively in the subsequent chunking phase, as TXT files might not contain natural delineations between topics or sections.

After extracting the textual content from each file type, normalization steps ensure a consistent and noise-free representation. These steps may include:

Whitespace and Line Break Management: Unnecessary whitespace, line breaks, and indentation are removed or standardized. This fosters uniformity in the final text and helps facilitate downstream segmentation.

Character Encoding Standardization: Ensuring all extracted text is encoded in a standardized character set (e.g., UTF-8) prevents issues in embedding and indexing stages.

Removal of Artifacts and Irrelevant Data: Headers, footers, or system-level strings like “Page X of Y” are stripped out to reduce the injection of irrelevant tokens. Similarly, artifacts such as timestamps or system-generated disclaimers, if not domain-relevant, are discarded to maintain a focus on essential content.

Managing Non-Textual Elements: While images, tables, and diagrams cannot be directly converted into text, associated captions or alternative text descriptions can sometimes be included. For documents where visual elements carry significant informational weight, a more advanced preprocessing might be considered, but for this thesis, the focus remains on textual data.

Throughout the preprocessing phase, utmost care is given to preserving domain-specific terminology and context. Academic environments at the University of Piraeus are replete with specialized jargon, unique course identifiers, departmental acronyms, and policy-specific language. Avoiding overly aggressive normalization steps—such as stemming or lemmatization at this stage—ensures that such domain-critical tokens remain intact. Retaining the original terminology improves semantic matching accuracy later, allowing the embeddings to capture the meaning and relevance of these specialized terms when users query the chatbot.

Once preprocessing completes, a quality assurance step verifies the integrity of the extracted text. Random sampling and inspection of a subset of documents ensure that no significant data corruption, truncation, or character encoding issues have occurred. If anomalies are detected—such as unintelligible text fragments or broken paragraphs—adjustments to parsing rules or filtering parameters can be made before proceeding with the embedding stage. This iterative approach ensures that data ingestion does not propagate errors into later stages of the pipeline.

The multi-format ingestion and preprocessing pipeline outlined here not only improves immediate retrieval and generative quality but also supports long-term maintainability and

scalability. As new documents are added or outdated files are replaced, administrators can re-run the preprocessing routines on fresh inputs. This modularity accommodates evolving academic requirements, ensuring that the chatbot remains current and authoritative as the university's information landscape changes. Moreover, the code-based implementation lends itself to automation, enabling scheduled or on-demand updates to incorporate newly published documents, revised departmental guidelines, or recently posted announcements without extensive manual intervention.

In the accompanying code, the function `process_documents()` systematically identifies each file within the configured `DOCUMENTS_FOLDER` and applies file-type-specific extraction logic. For PDFs, the script uses `PyPDF2` (`extract_text_from_pdf`) to read page-by-page text layers, while for DOCX files it employs `python-docx` (`extract_text_from_word`) to iterate through document paragraphs. Tabular data in CSV or XLSX format is processed by `pandas` (`extract_text_from_tabular`) to convert rows and columns into a human-readable string. Finally, `.txt` files are loaded directly in UTF-8 encoding. These specialized helper functions ensure that each source—whether an administrative guideline in PDF or a spreadsheet of course information—undergoes appropriate parsing prior to subsequent chunking and embedding.

3.1.2 Chunking Text and Generating Semantic Embeddings (SentenceTransformers)

Effective integration of domain-specific knowledge into a Retrieval-Augmented Generation (RAG) architecture requires a granular, semantically meaningful representation of the textual corpus. Following the normalization and preprocessing steps described in the previous section, the text now exists in a consistent and noise-free format. However, even well-prepared textual data often remains too extensive or heterogeneous for direct use in retrieval pipelines. Long, unsegmented documents can obscure topical boundaries and make it difficult to align user queries with the most relevant information. To address these challenges, a strategic “chunking” process is employed to divide texts into manageable, self-contained segments, which are then transformed into dense semantic embeddings using a specialized embedding model, such as `SentenceTransformers`.

Chunking involves splitting each document into smaller pieces, commonly consisting of a few hundred words. This segmentation strategy enables a more fine-grained retrieval process. Rather than matching a user query against an entire multi-page document, the system can pinpoint relevant information in a localized chunk that closely aligns with the query's intent. The benefits of this approach are twofold:

When dealing with broad, multi-topic documents, queries might correspond to specific sections of text rather than the entire file. By chunking text into narrower segments (e.g., 300–500 words), the system reduces the semantic noise that could misdirect retrieval mechanisms. For instance, a single policy document might cover admissions procedures, graduation requirements, and faculty governance. Without chunking, a query about “graduate thesis deadlines” could return large segments containing irrelevant details. With chunking, the retrieval engine can zero in on the chunk discussing submission guidelines, improving accuracy and relevance.

During the prompt construction phase, only a handful of the most relevant chunks are included as context for the language model. Smaller, self-contained chunks are easier to manage in terms of token budgeting, ensuring that the model's context window is not overwhelmed by large swaths of text. This targeted inclusion prevents unnecessary token consumption and preserves space for user input and other essential components of the prompt.

Selecting an optimal chunk size is a balance between maintaining semantic coherence and ensuring manageable granularity. Too large a chunk risks encompassing multiple unrelated topics, while too small a chunk may fragment coherent information streams, making it harder for the model to grasp context. A commonly used heuristic is to base chunk size on the number of words. For this project, a chunk size of approximately 500 words proved effective, as it generally captures a substantive conceptual unit—such as a subsection of a course syllabus or a block of policy text—while remaining compact enough for precise retrieval.

In practice, the optimal chunk size may vary depending on the domain and the nature of the documents. Highly structured documents (e.g., academic regulations with clearly defined sections) might permit larger chunks, whereas more free-form narrative texts (such as conference announcements or department news) might benefit from slightly smaller segments. The choice of chunk size can be re-evaluated iteratively as system performance and user feedback guide adjustments.

Once the text has been chunked, each chunk is transformed into a high-dimensional vector representation—or embedding—that captures semantic relationships and contextual nuances. For this task, SentenceTransformers (Reimers & Gurevych, 2019) is utilized. SentenceTransformers provides a flexible and efficient way to generate embeddings for sentences, paragraphs, and documents, leveraging models pre-trained on large corpora and often fine-tuned on specific semantic similarity tasks.

SentenceTransformers models excel at capturing the meaning and semantic relationships within text. Rather than relying solely on surface-level features, these embeddings reflect deeper conceptual linkages. For a specialized academic environment like the University of Piraeus, this capability is crucial. Embeddings allow the system to understand that “departmental thesis guidelines” and “graduate thesis submission deadlines” are conceptually related, improving the accuracy of retrieval and generation.

The generated embeddings are typically high-dimensional vectors. When user queries are similarly embedded, measuring their cosine similarity to stored chunk embeddings enables the system to identify which chunks are most pertinent. These similarity calculations are computationally efficient and work well at scale. This allows for rapid retrieval across large document repositories, even as new materials are continually added.

SentenceTransformers supports various pre-trained models. For a general academic domain, a well-established model like “paraphrase-multilingual-MiniLM-L12-v2” can be chosen to handle diverse linguistic structures and terminologies. Should the domain shift or become more specialized, models fine-tuned on domain-relevant corpora can be easily integrated. This adaptability ensures the system remains flexible in addressing evolving academic content and user demands.

The embeddings generated by SentenceTransformers inherently capture a wide range of textual semantics, but domain-specific terms may require special consideration. Although no explicit domain-adaptation is performed at this stage, the model's capability to represent words and phrases within their given contexts ensures that recurrent academic terms, course codes, departmental acronyms, and policy keywords gain meaningful vector representations. Over time, as queries consistently return certain chunks in response to particular domain topics, these specialized embeddings become even more valuable in guiding precise and context-rich responses.

As with document preprocessing, the embedding generation process benefits from periodic quality checks. Randomly selecting chunks and examining their top semantic neighbors can help validate that the embeddings encode coherent relationships. For example, a chunk describing “graduate program admission requirements” should retrieve segments

related to admissions, graduate studies, or registration deadlines rather than unrelated content. If anomalies occur—such as embeddings that cluster semantically unrelated chunks—additional fine-tuning, normalization, or model selection adjustments may be warranted.

Following the embedding generation phase, these dense vector representations are stored alongside their corresponding textual chunks in a database (as detailed in Section 3.1.3). The seamless interaction between stored embeddings and the retrieval module (Section 3.2) allows the system to quickly filter, rank, and select top-k relevant chunks for any given user query. This pipeline ensures that when a user asks a specialized question—such as “What are the prerequisites for the master’s program in Data Analytics?”—the system can swiftly reference the correct segments of related documents, providing a factually grounded context to the language model.

The code introduces a `chunk_text` utility that segments large bodies of text into substrings of approximately 500 words, which aligns with the design choice discussed earlier. Immediately after chunking, the script invokes `embedding_model.encode(chunks, convert_to_tensor=False)` to transform each chunk into a dense vector using `SentenceTransformers`. Notably, the snippet then converts these vectors to `numpy.float32`—a memory-efficient format—before proceeding with database insertion. By encoding chunks in this manner, the system captures nuanced semantic relationships, allowing queries to be matched with the most relevant passages later in the retrieval phase.

3.1.3 Storing Embeddings and Metadata in MongoDB

Once the text has been pre-processed, chunked, and transformed into semantically meaningful embeddings, the next step involves efficiently organizing and storing these results to support rapid retrieval and scalable system maintenance. MongoDB, a NoSQL document-oriented database, serves as the backbone for persisting the embeddings, associated textual content, and relevant metadata. By adopting a flexible and schema-less storage model, MongoDB allows for straightforward updates to the corpus, efficient indexing strategies, and smooth integration with downstream components of the Retrieval-Augmented Generation (RAG) pipeline.

Traditional relational databases, while highly structured, can be cumbersome when handling heterogeneous data that may evolve over time. The academic domain, such as that of the University of Piraeus, often deals with evolving documentation—new guidelines, updated course offerings, revised policies—that require frequent insertions and modifications. MongoDB’s document-oriented model is particularly well-suited to this dynamic environment. Each chunk of text, along with its embedding and metadata, can be stored as a single document. This reduces the impedance mismatch between the programmatic data structures used during preprocessing and the storage layer, streamlining both code complexity and development overhead.

In addition, MongoDB’s indexing capabilities offer robust performance benefits. By creating indexes on fields like `filename`, `chunk index`, or other key attributes, queries can run efficiently, even as the dataset grows. This efficiency is critical for ensuring that retrieval steps remain responsive, minimizing latency for end-users interacting with the chatbot.

Although MongoDB is schemeless, establishing a logical schema at the design stage promotes consistency and ease of maintenance. A typical document in the database includes the following fields:

- **filename:** A string representing the source document's filename (e.g., "admissions_policy.pdf"). This metadata helps trace each chunk back to its original source, enabling transparency and trust in the information provided by the chatbot.
- **chunk_index:** An integer indicating the chunk's position within the parent document. By maintaining this index, updates or removals of specific chunks are simplified, and it becomes easier to navigate or reconstruct context if needed.
- **text:** The cleaned and normalized textual content of the chunk. Storing the text alongside the embedding ensures that if embeddings need to be regenerated or validated, the original content remains readily accessible. In addition, providing the exact text improves auditing and debugging processes, as developers and administrators can inspect the precise content that led to a given model response.
- **embedding:** A list (or array) of floating-point values representing the semantic embedding of the text chunk. Typically, these embeddings are high-dimensional vectors (e.g., 384 or 768 dimensions, depending on the chosen model). Although high-dimensional arrays can consume significant storage space, MongoDB's document structure and the compression strategies available help manage storage overhead. Storing embeddings as arrays of floats facilitates straightforward retrieval and similarity computations when these vectors are loaded back into memory for query processing.

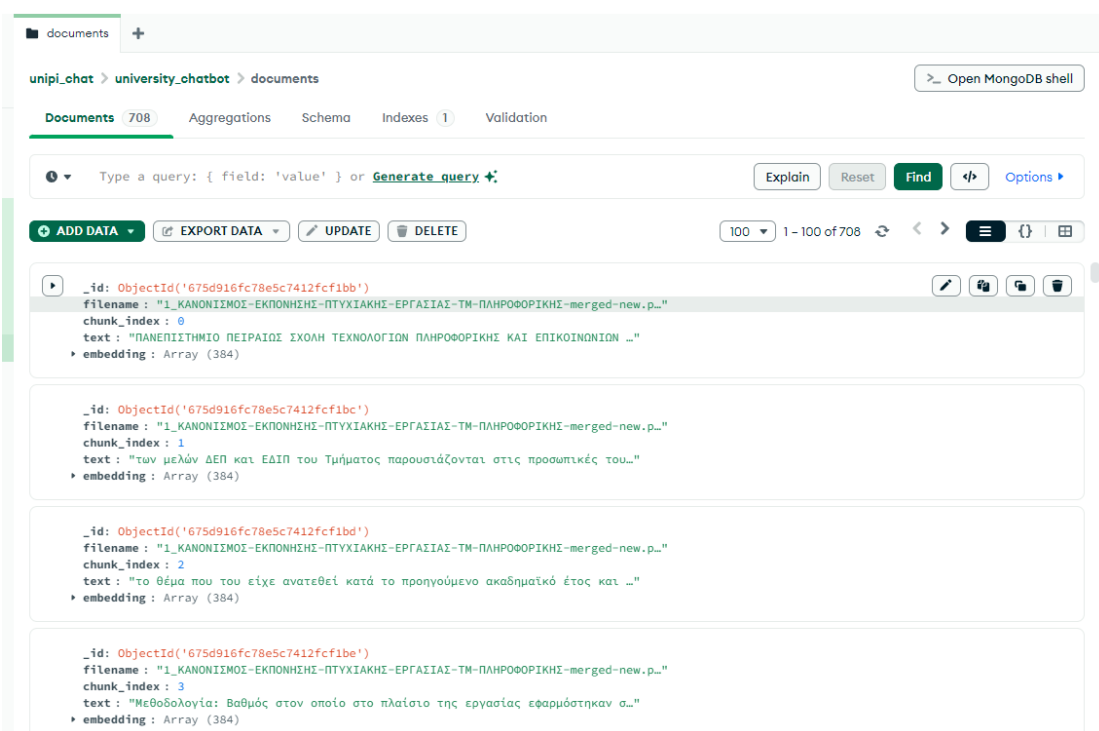


Figure 3. The database

The process of inserting documents into MongoDB typically occurs after the embedding generation phase. For each chunk produced and embedded, the system constructs a corresponding MongoDB document. A bulk insertion operation can then efficiently write thousands of documents at once, minimizing round-trip times and reducing overhead.

Over time, as the university updates its policies or introduces new documents, incremental insertions can refresh the corpus without disrupting the existing knowledge base. If a particular document changes, only the affected chunks and their embeddings need to be re-processed and updated in the database. This modularity ensures that the system remains

consistently aligned with the latest information—an essential feature in dynamic academic settings where policies and procedures evolve regularly.

To facilitate rapid retrieval, indexes can be established on frequently queried fields. For instance, creating an index on the “filename” field allows administrators to quickly isolate all chunks related to a specific source document. Although embeddings themselves are not directly indexed by MongoDB in their native vector form, their selection and ranking occur after these vectors are loaded into memory and processed by a similarity search algorithm (detailed in Section 3.2.1). However, metadata-based indexing can still improve the efficiency of pre-filtering steps, such as narrowing the search to documents of a particular type (e.g., policy documents vs. course syllabi) before conducting a similarity-based retrieval.

One of the key advantages of leveraging MongoDB is the ability to scale horizontally by distributing data across multiple nodes via sharding. As the number of documents—i.e., chunks—increases with the university’s growing corpus, shards can be added to maintain performance. This scaling flexibility ensures that the chatbot can handle increasing query volumes and larger collections of academic content without substantial architectural overhauls.

Backup and restoration procedures, data migrations, and integration with continuous deployment pipelines are also facilitated by MongoDB’s ecosystem. By following best practices—such as regular backups, monitoring disk usage, and reviewing query performance—developers can maintain a stable and responsive data layer that underpins the RAG-based chatbot’s retrieval capabilities.

Despite MongoDB’s flexibility, careful data governance remains important. Routine checks can confirm that embeddings and text remain synchronized, that no duplicate chunks exist due to accidental repeated insertion, and that newly added documents maintain the required quality standards. In cases where duplicates or inconsistencies are detected, administrators can apply MongoDB’s querying and updating functionalities to clean or remove affected documents.

Moreover, as embedding models or chunking strategies evolve, re-running the entire ingestion pipeline and re-inserting updated documents can be done seamlessly. By retaining both old and new embeddings for a transition period, developers can experiment with improved embedding models without immediately discarding previously stored information. This iterative refinement ensures that the chatbot’s knowledge base can continuously improve in accuracy, semantic clarity, and retrieval efficiency.

After generating embeddings, the script checks for duplicates in MongoDB by matching both the filename and the exact text chunk. If a match is found, the pipeline skips insertion to avoid redundant entries. This document is then inserted into the MongoDB collection. Storing embeddings and metadata side by side ensures straightforward retrieval and inspection. For instance, if a user’s query requires referencing a specific policy, the system can trace the relevant embedding back to its parent file chunk. The `process_documents()` function thus provides a clear automation workflow: read file contents, chunk and embed, skip duplicates, then persist each unique chunk along with its embedding in MongoDB.

3.2 Retrieval and Clustering Module

The Retrieval and Clustering Module serves as the operational core that bridges the static knowledge base of embeddings and text chunks with the dynamic, query-driven demands of the user. At runtime, the chatbot must rapidly sift through a large corpus of semantically encoded chunks, identify those most relevant to the user’s query, and present them in a coherent manner to the generative component. To achieve this, the retrieval process is

structured into two key phases: an initial similarity search against the stored embeddings, and a subsequent clustering-based refinement that enhances contextual alignment. This two-step approach ensures that the chatbot not only finds topical matches but also constructs a semantically cohesive context that improves the clarity and factual grounding of its eventual responses.

3.2.1 Vector Similarity Search for Domain-Specific Content

The first step in retrieval leverages the embeddings generated and stored in the MongoDB database. When a user submits a query, the system encodes it into an embedding vector using the same SentenceTransformers model employed during data ingestion. This ensures consistent vector representations, allowing for a straightforward comparison between the query vector and the vectors associated with each text chunk. Cosine similarity, a common measure of vector proximity, is used to identify the top-k chunks whose semantic content most closely aligns with the query.

This embedding-based retrieval method offers several advantages over keyword-based approaches. By relying on semantic similarity, the system can recognize that a query about “Master’s thesis deadlines” is related to content discussing “graduate research submission timelines,” even if the exact keywords do not match. In a domain like a university setting—where official documents may use formal terms, while users may pose queries in a more colloquial style—this flexibility is critical. Moreover, the embeddings capture context-rich semantic fields, enabling the retrieval mechanism to rank passages not merely by keyword overlap, but by conceptual relevance.

The similarity search phase involves loading embeddings from the database into memory, computing similarities to the query vector, and then selecting the top-k candidates. Although the number k may be tuned, values in the range of 10 to 50 are common starting points, ensuring that a sufficient breadth of potentially relevant chunks is retained for further refinement. The goal of this initial retrieval step is recall-oriented—i.e., to gather a set of context candidates that likely contain the answer or relevant information. In doing so, it sets the stage for a subsequent clustering-based filtration and refinement step.

In practice, once the top-k chunks are retrieved, they are passed—along with their embeddings and similarity scores—to a clustering function (`clean_and_reconstruct_context`) which further refines the context before final prompt construction, as described in the subsequent subsection (3.2.2).

3.2.2 K-Means Clustering for Context Refinement

Once the initial set of top-k chunks is obtained, the system applies a clustering step to further refine and organize this candidate pool. Traditional k-nearest neighbour retrieval strategies might return multiple similar or redundant chunks that add little distinct value to the final prompt. By contrast, clustering techniques help diversify the retrieved content, ensuring that each piece of included context contributes unique information, rather than reiterating the same points.

K-means clustering, a widely adopted and computationally efficient algorithm, is particularly well-suited for this purpose. Applied to the embedding vectors of the top-k retrieved chunks, k-means partitions them into a specified number of clusters (e.g., `n_clusters = 5` or `10`). Each cluster represents a thematic subset of the candidate documents. Because chunks are grouped based on their semantic proximity, clusters naturally align with latent topics or subtopics related to the user’s query.

In the provided implementation, the `clean_and_reconstruct_context()` function demonstrates how top-k retrieval results are grouped using k-means clustering. Each text chunk, along with its similarity score, is analyzed to ensure that redundant or near-duplicate chunks are filtered out. This step leverages the scikit-learn KMeans class, as shown in the code snippet.

```
def clean_and_reconstruct_context(results, top_k=10, n_clusters=5):
    """
    Remove common texts from results and replace them using k-means clustering.

    Args:
        results (list of tuples): List of (text, similarity score) tuples.
        top_k (int): Number of top chunks to keep in context.
        n_clusters (int): Number of clusters for k-means.

    Returns:
        str: Cleaned and reconstructed context.
    """
    # Extract texts and scores
    texts = [text for text, _ in results]
    embeddings = np.array([embedding for _, embedding in results]) # Assuming embeddings
are in results

    # Ensure embeddings are 2D
    embeddings = np.atleast_2d(embeddings)

    # Adjust the number of clusters if there are fewer samples than clusters
    n_clusters = min(n_clusters, len(embeddings))

    if n_clusters > 1:
        # Apply k-means clustering
        kmeans = KMeans(n_clusters=n_clusters, random_state=42)
        cluster_labels = kmeans.fit_predict(embeddings)

        # Group texts by cluster
        clustered_texts = {}
        for idx, label in enumerate(cluster_labels):
            if label not in clustered_texts:
                clustered_texts[label] = []
            clustered_texts[label].append((texts[idx], results[idx][1])) # (text, similarity score)
```

```
# Select top texts from each cluster
unique_texts = []
for label, chunks in clustered_texts.items():
    # Sort chunks in the cluster by similarity score
    chunks = sorted(chunks, key=lambda x: x[1], reverse=True)
    unique_texts.append(chunks[0][0]) # Take the most relevant text from each cluster

# If fewer unique texts than required, fill with the next most relevant chunks
remaining_texts = [text for text in texts if text not in unique_texts]
while len(unique_texts) < top_k and remaining_texts:
    unique_texts.append(remaining_texts.pop(0))
else:
    # If only one cluster, use the top_k texts directly
    unique_texts = texts[:top_k]

# Reconstruct the context
context = "\n".join(unique_texts[:top_k])
return context
```

This clustering step benefits the system in several ways:

Reducing Redundancy:

By examining the resulting clusters, the system can select only the top representative chunks from each cluster. Instead of overwhelming the prompt with multiple near-duplicate chunks, it includes only the most relevant piece per cluster. This ensures the generative model receives a more balanced, thematically diverse context, leading to richer and more precise responses.

Enhancing Contextual Density:

Clustering highlights the internal structure of the candidate set. If a query about “departmental funding opportunities” retrieves numerous passages all related to research grants, the clustering process will group them, making it easier to pick the one chunk that best summarizes that aspect. If another cluster covers scholarships and a third details administrative procedures, the final selected set of chunks will collectively offer a broad, well-rounded context tailored to the user’s needs.

Efficient Token Management:

With fewer redundant chunks, the final context passed to the generative model consumes fewer tokens, optimizing the token budget. This token efficiency is especially important given the constraints of large language model APIs. A carefully chosen representative from each cluster ensures that the generative model receives the most information-rich content without unnecessary repetition or filler text.

Adaptive to Corpus Expansion:

As new documents are ingested and more embeddings are stored, the distribution of candidate chunks returned for a given query may shift. Clustering dynamically adapts to this changing landscape. With each retrieval operation, the chosen clusters form anew, reflecting the current structure of the corpus. This adaptability ensures that the chatbot can remain responsive and maintain high-quality retrieval performance as the university's knowledge base evolves over time.

As illustrated in the `clean_and_reconstruct_context()` function, a default `n_clusters` value (e.g., 5 or 10) is applied. If the system finds fewer chunks than the intended number of clusters, it automatically adjusts `n_clusters` to match the available data. Once clustering labels are assigned, the code picks the chunk with the highest similarity score within each cluster, ensuring that only the top representative text is included. The remaining chunks serve as potential backups if additional context is required. This approach balances diversity and token efficiency, enabling a well-rounded final context for the generative model.

Implementation Details and Parameter Tuning:

In practice, implementing k-means clustering involves specifying parameters such as the number of clusters (`n_clusters`) and initialization methods. While a default choice of `n_clusters` = 5 or 10 may suffice initially, empirical testing can guide adjustments. More clusters may foster granularity but risk overfragmentation, while fewer clusters may yield insufficient thematic diversity.

Moreover, the selection criteria for choosing the top representative chunk from each cluster can vary. One common strategy is to pick the chunk with the highest similarity to the user query within each cluster. Another approach might weigh additional factors, such as document recency or chunk length, depending on the system's domain requirements and performance goals.

The retrieval and clustering module's output—a refined set of top chunks—is passed to the token management and prompt construction stage (Section 3.4). At that point, the best candidates are incorporated into the prompt provided to the OpenAI GPT API. Ensuring a clean and semantically dense context sets the stage for the generative model to produce accurate and contextually grounded responses to user queries about the University of Piraeus.

Moreover, as the system undergoes continuous use, logs of retrieval and clustering results can be examined to improve performance. If certain queries consistently yield suboptimal clusters or fail to address user needs adequately, developers can revisit parameter choices, adopt more advanced clustering techniques, or refine chunking and embedding strategies in earlier stages.

Finally, `clean_and_reconstruct_context()` reconstructs the context by merging the selected representative chunks—up to a specified limit (`top_k`)—into a single string. This condensed, thematically diverse context is then passed to the language model via the Flask endpoint (`/query`). Hence, k-means clustering and the subsequent filtering mechanism play a pivotal role in ensuring that the chatbot returns high-quality, domain-specific answers.

3.3 Generative Component for Specialized Chatbot Generation

With the corpus of domain-specific documents processed, embedded, and refined through retrieval and clustering, the final step in producing a coherent and contextually accurate response rests on the generative component. This segment of the pipeline synthesizes the user query and retrieved context into a well-structured, human-like answer, leveraging the

capabilities of a large language model (LLM) operating under the Retrieval-Augmented Generation (RAG) paradigm. By integrating carefully selected context chunks with state-of-the-art generative technologies, the system aims to produce responses that are both factually grounded and linguistically fluent.

3.3.1 Integration with OpenAI GPT API

The generative engine in this implementation is based on the OpenAI GPT API, offering advanced text generation capabilities and robust support for prompt-based conditioning (Brown et al., 2020). Models like GPT-3.5 have demonstrated remarkable linguistic fluency and versatility across a wide range of domains. However, when operating in specialized contexts—such as responding to queries about university policies, courses, or administrative details—purely generative models may inadvertently produce hallucinations or fabricate content beyond their training data (Ji et al., 2023). To counteract these tendencies, the RAG architecture supplies a curated context set drawn from the domain’s knowledge repository, thereby anchoring the model’s reasoning in verifiable information.

During inference, the user’s query is combined with the top-ranked and cluster-refined chunks retrieved from the MongoDB database. This fused prompt typically includes:

A high-level system instruction or role specification (e.g., “You are a knowledgeable assistant for the University of Piraeus...”), providing a guiding framework for the model’s persona and output style.

The user’s query, ensuring the model remains focused on the current informational need.

Several context chunks selected from the retrieval and clustering module. These textual segments may represent course requirements, departmental regulations, faculty profiles, or procedural instructions, all directly sourced from the university’s official corpus. By integrating these segments, the generative model receives accurate, domain-aligned background information, thus improving the factual correctness of its output.

The OpenAI GPT API’s adjustable parameters (e.g., temperature, top-p sampling) enable fine-tuning the model’s creativity and risk of producing off-topic responses. Since academic and institutional queries generally demand precision over stylistic flair, these parameters are typically set to lower creativity values, guiding the model to lean more heavily on the provided context rather than speculative reasoning. In this manner, the generative component functions less as a free-form writer and more as a contextually aware advisor, reflecting the ethos of specialized chatbot deployment (Lewis et al., 2020).

In the actual implementation, the Flask endpoint (/query) orchestrates the entire retrieval-generation pipeline. First, it invokes `query_similar_chunks` to gather top-ranked chunks, applies `clean_and_reconstruct_context` for clustering and filtering, and then carefully trims the final context if it exceeds a token threshold (see `max_context_length = 1000`). Finally, it appends the user’s query and the refined context to a prompt, which is dispatched to `query_gpt` for OpenAI GPT inference. This sequence ensures that the model’s generative output is grounded in domain-specific information without overwhelming the token limits.

3.3.2 Balancing Fluency, Accuracy, and Domain Relevance

The success of a RAG-based specialized chatbot hinges on achieving a balance between the model’s inherent fluency and the user’s need for precise, domain-specific answers. Generative models, by design, can produce remarkably coherent and human-like text. Yet, within academic

or administrative settings, mere fluency is insufficient. The chatbot's output must adhere to the factual constraints set by the retrieved documents, mirror appropriate institutional tone, and withstand scrutiny from knowledgeable users.

By providing thematically aligned chunks, the RAG system constrains the model's generative space. Rather than relying on latent knowledge embedded in model parameters, the model references explicitly supplied content. Studies in retrieval-augmented architectures highlight that grounding generation in retrieved evidence substantially improves factual accuracy and reduces the incidence of hallucinations (Lewis et al., 2020; Shuster et al., 2022). For instance, if a user asks for the required credits to graduate from a certain master's program, the retrieved chunks may detail the exact number of credits, their distribution across core and elective courses, and any thesis requirements. The model, guided by these contextual details, can confidently present the correct information in a cohesive narrative.

To maintain continuity and a sense of conversational flow, the system appends both the user query and GPT responses to a `chat_history` list. This short-term memory (last 3 turns) is incorporated into subsequent prompts. The final GPT output thus remains both contextually focused and mindful of recent exchanges, striking a balance between academic precision and conversational fluency.

3.3.3 Prompt Construction and Token Efficiency

Effective prompt construction is integral to the generative component's success. Section 3.4 elaborates on token management and how the system optimizes prompt length to fit within the model's context window (Liu et al., 2021). Here, it is sufficient to note that the final prompt merges carefully selected text segments, filtered by semantic relevance and clustering diversity, into a structured input. Typically, the prompt might follow a pattern similar to:

System Role/Instruction: A brief directive establishing the assistant's persona and objectives (e.g., "You are a helpful assistant providing authoritative and accurate information about the University of Piraeus's academic offerings, rules, and policies.").

Contextual Segments: A concatenation of 3–7 (depending on token constraints) of the most relevant, non-redundant chunks retrieved from the corpus. Each chunk can be preceded by a short descriptor or heading if needed, though often raw concatenation suffices.

User Query: The user's original question, placed either after or before the contextual segments depending on best practices and empirical performance. Some configurations present the user query at the end of the prompt to ensure the model has immediate reference to the specific request, while others place it earlier to highlight the main user goal.

Since the available token window for GPT models is finite, the number and size of included chunks may vary according to the complexity of the query and the richness of the underlying corpus. Adaptive methods can be employed to dynamically select shorter chunks or summarize multiple chunks into a compact form if token limits become restrictive. The ultimate aim is to present the model with sufficient information to answer the query accurately without sacrificing fluency or incurring excessive computational costs.

In practice, the `/query` route sets `max_context_length = 1000` to approximate the token capacity for GPT input. If the reconstructed context—after retrieval and clustering—surpasses this limit, the system logs a warning ('Trimming context due to token limit.') and truncates the text. This safeguard prevents the final prompt from exceeding the model's window and incurring incomplete or error-prone responses. By carefully monitoring and trimming tokens, the code ensures that user queries, domain-specific content, and minimal instruction overhead are included in the final prompt.

3.3.4 Handling Ambiguity and Uncertainty

In specialized domains, it is not uncommon for user queries to be ambiguous or for the provided corpus to lack a definitive answer. Under these circumstances, the generative component must maintain academic integrity and transparency. Rather than fabricating information, the model should gracefully acknowledge uncertainty or advise users on where to seek clarifications.

To encourage such behavior, the system's prompt can instruct the model to refrain from speculation and to base answers strictly on the provided context. If no relevant chunks are found or if the retrieved documents do not contain the requested information, the model can be guided to respond with a neutral acknowledgment (e.g., "I'm sorry, but I don't have information on that topic.") or suggest alternative sources. Such guardrails help uphold the chatbot's credibility and align it with ethical and academic standards (Zhang et al., 2022).

When encountering ambiguous queries or insufficient context, the `/query` function integrates a fallback behavior: it logs the limited context, includes it in the prompt, and instructs the GPT model to respond with caution. If no relevant chunks are found, the system can return a neutral response or indicate that more information is needed, preserving the chatbot's credibility and academic integrity.

3.3.5 Continuous Improvement and Model Selection

The generative component's performance may evolve over time. As improved LLM variants become available, system maintainers can switch to newer models with larger context windows, better reasoning capabilities, or domain fine-tuning. Moreover, analysing user feedback, logs, and error cases can inform adjustments to prompt templates, retrieval strategies, and token management policies. Over iterative deployments, the generative component may be tuned to better reflect the evolving landscape of university regulations, course offerings, and institutional priorities.

Experimental benchmarks—such as simulated user queries comparing responses generated before and after system refinements—can assess the effectiveness of changes. Additionally, structured user studies involving faculty, administrators, and students can provide qualitative feedback. This feedback loop ensures that the generative component aligns closely with the dynamic, domain-specific needs of the University of Piraeus community, consistently delivering reliable and contextually grounded answers.

Additionally, each query and response is printed to the console (`print(chat_history)`) for basic observability. Over time, developers can extract these logs, analyze common user questions, and refine the retrieval or prompt construction strategies based on observed issues, thereby continually enhancing the chatbot's domain alignment and factual correctness.

3.4 Token Management and Prompt Construction

The process of guiding a large language model (LLM) to produce accurate, contextually aligned responses is heavily influenced by how information is presented in the input prompt. Modern LLMs, such as those accessed through the OpenAI GPT API, operate within predefined context window limits—commonly ranging from a few thousand to tens of thousands of tokens. Ensuring that essential information fits into this limited space without sacrificing clarity, factuality, or user relevance is a key challenge. Token management and prompt construction techniques seek to address this issue by carefully selecting, organizing, and trimming the prompt content.

3.4.1 Importance of Token Constraints and Costs

Token management is not merely a technical detail; it influences the system's usability, responsiveness, and operating costs. Exceeding token limits can cause the model to truncate the input, omit critical context, or fail to generate a coherent answer. Even when staying within these limits, unnecessary token consumption leads to higher inference costs, increased latency, and potentially reduced service scalability.

In specialized domains, such as the University of Piraeus environment addressed in this system, queries may require referencing multiple documents or large segments of text. Overly lengthy or redundant prompts risk overwhelming the model's context window. Therefore, token management strategies—such as content prioritization, careful truncation, and token-efficient formatting—enable the chatbot to remain agile and cost-effective while maintaining response quality.

In this system, the Flask endpoint (/query) demonstrates how token constraints are enforced in practice. Once retrieval and clustering have produced a context string, the code checks if `len(context.split()) > max_context_length` (with `max_context_length = 1000`). If exceeded, the context is truncated to fit within this limit. This ensures that user queries remain both cost-efficient and responsive, mitigating incomplete or error-prone answers caused by overlong prompts.

3.4.2 Selecting Relevant Chunks

The retrieval and clustering processes described in Section 3.2 aim to produce a refined set of thematically pertinent document segments. However, the number of retrieved chunks often exceeds the practical limits of the LLM's context window. To optimize token usage, the system prioritizes content based on semantic relevance, diversity, and potential informational value.

Additionally, the system retrieves up to 50 chunks (`top_k=50`) before clustering or token trimming. By capping the number of chunks in the final context, we ensure that only the most valuable segments survive the subsequent filtration and token checks, thereby reducing redundancy and token overconsumption.

A common approach involves scoring each chunk by its similarity to the user's query and then including only the top-scoring ones. If the chunk set remains too large, additional heuristics can be applied—such as favouring shorter but information-rich segments, or preferring chunks that have not been included in recent queries (to avoid repetitive contexts). The final selection ensures that the model receives a concise, high-density context package, thereby increasing the odds that it can address the user's question comprehensively within the allotted token budget.

3.4.3 Hierarchical and Layered Prompt Designs

When users pose complex queries requiring multiple layers of context, hierarchical prompt construction can prove beneficial. Instead of feeding all relevant chunks directly into the final prompt, the system may employ an intermediate summarization step. For example, it could first prompt the model (or a separate summarization tool) to condense several related chunks into a shorter, high-level summary. This summary then replaces the original chunks in the final prompt presented to the LLM. By converting numerous detailed segments into a single, well-structured summary, the system conserves tokens without discarding essential information.

This two-stage approach—retrieval and summarization, followed by final query response generation—balances granularity with compactness. Summarization may also unify the narrative, making it easier for the model to understand the overarching context and respond with a coherent, authoritative answer.

3.4.4 Adaptive Truncation Techniques

Even after careful chunk selection and optional summarization, prompts may still exceed desirable token limits, especially when dealing with large corpora or particularly complex queries. Adaptive truncation algorithms dynamically adjust the amount of included text based on the evolving token count.

These techniques can start by including the most critical chunks first, then progressively add less essential content until reaching the token boundary. If the token count approaches the limit, the system truncates from the least relevant end—often discarding tangential or repetitive segments. Such adaptivity ensures that each query receives an optimally tailored prompt, responsive to the uniqueness of the user's request and the constraints of the model.

In the actual /query implementation, after chunk selection and clustering, the code explicitly trims the final context if it exceeds the `max_context_length`. By performing this truncation in a single pass, the system retains critical information at the beginning of the prompt while discarding lower-priority text near the end, maintaining coherence and relevance within a strict token budget.

3.4.5 Maintaining Coherence and Logical Flow

While efficiency and token constraints are key drivers of prompt construction, the final prompt must also maintain logical coherence and readability. LLMs tend to produce better outcomes when the prompt is structured and comprehensible. Formatting choices—such as clear headings, delineated sections, or simple bullet points—can help the model navigate the provided information more effectively.

For instance, the prompt might include a section labeled “Context,” followed by the selected chunks separated by line breaks or markdown headings, and then a section labeled “User Query,” where the user's exact question is presented. Minimally, a well-labeled prompt reduces ambiguity and directs the model's attention to the appropriate parts of the input.

To further preserve logical flow, the code merges selected chunks into a single text block labeled ‘Πλαίσιο.’ (Context) in Greek, followed by the user query and recent conversation history, thereby helping the language model differentiate between background context, user intent, and preceding dialogue.

3.4.6 Balancing Domain Specificity and General Guidance

Domain-specific detail is the hallmark of a specialized chatbot, but the prompt can also include general instructions that shape the model's response style. For example, stating “Respond accurately and concisely using the provided university policies” or “If uncertain, indicate the need for more information” can guide the model's behavior. However, these meta-instructions also count as tokens. Efficient token management involves striking a balance: including enough guidance to steer the model toward domain-appropriate behavior without consuming excessive space that could be reserved for critical context chunks.

3.4.7 Iterative Refinement and Evaluation

Optimal prompt construction and token management policies rarely emerge fully formed. Instead, they are refined iteratively based on empirical feedback. By analyzing logs of model responses, user satisfaction metrics, and error cases, developers can adjust the heuristics for chunk selection, tune the maximum token thresholds, or experiment with new summarization strategies.

A systematic evaluation cycle might involve:

- **A/B Testing:** Presenting the model with the same query under different prompt constructions and comparing response quality.
- **User Feedback:** Gathering input from students, faculty, and staff to identify whether the chatbot is providing too much irrelevant detail, missing important nuances, or displaying inconsistent accuracy.
- **Performance Benchmarks:** Measuring response latency and token consumption across multiple queries to ensure that prompt construction scales well under varying loads and corpora sizes.

Through iterative refinement, the system learns which configurations yield the best trade-offs between completeness, accuracy, and brevity, ultimately improving the user experience.

3.4.8 Future Directions in Token Management

As LLMs evolve and offer larger context windows, some token management challenges may diminish. However, even with expanded capacities, efficient prompt construction will remain valuable. More complex requests, richer domain data, and a proliferation of relevant documents may still threaten to overwhelm input limits. The evolving sophistication of RAG-based architectures and emerging techniques—such as dynamic retrieval, segment embeddings, or advanced compression models—will likely inform future token management strategies.

Moreover, integrating personalized features, user history, or more nuanced retrieval filters could make token management even more critical, as these additions increase complexity. Successful token management ensures that the system can handle growth in both data size and user expectations without sacrificing accuracy or performance.

3.5 Backend Implementation (Flask API) and Services Integration

The backend layer serves as the orchestrating engine that connects frontend user requests with the underlying RAG-based architecture, embedding storage, retrieval strategies, generative components, and token management routines. Implemented using Flask—a lightweight and flexible Python web framework—the backend abstracts the complexity of data pipelines, model inference, and prompt engineering into a set of well-defined RESTful endpoints. Through these interfaces, the frontend can submit queries, obtain contextually enriched responses, and maintain conversation histories without direct exposure to the intricate mechanisms that drive the chatbot's behavior.

3.5.1 Flask as the Integration Hub

Flask is chosen for its simplicity, modularity, and extensive ecosystem of extensions and libraries, making it well-suited for rapid prototyping and seamless integration with external services. Its minimalist design allows developers to focus on implementing the core logic and data handling workflows that align with the RAG-based approach, rather than navigating a monolithic framework. In this implementation, the backend code is organized into blueprints

and services, maintaining clear boundaries between various functional components—such as retrieval, embedding management, and query handling.

A primary example is the `query_blueprint`, which defines the `/query` route responsible for receiving user inquiries, orchestrating retrieval and clustering, enforcing token limits, and dispatching prompts to the OpenAI GPT API. This design ensures that changes in one component, such as adjusting chunk selection heuristics or modifying the token trimming policy, can be made without disrupting other modules.

3.5.2 Core Responsibilities of the Backend

The backend undertakes several key responsibilities in the RAG pipeline:

1. Request Handling and Input Validation:

When a user submits a query through the React-based frontend, the request arrives at a dedicated Flask endpoint (`POST /query`). The backend validates the incoming payload, verifying that essential parameters—such as the user’s question—are present and correctly formatted. By performing these checks early, the backend prevents malformed inputs or empty queries from propagating downstream.

2. Retrieval-Oriented Workflows:

Leveraging the retrieval functions discussed in Section 3.2, the backend obtains the most semantically relevant chunks from MongoDB, computing cosine similarity between the user’s query embedding and stored document embeddings. This step may yield up to 50 top-ranked chunks (`top_k=50`). To reduce redundancy and promote contextual diversity, the backend then applies a clustering function (`clean_and_reconstruct_context`). The refined set of chunks emerges from k-means clustering, which removes near-duplicates and consolidates thematically aligned segments.

3. Contextual Prompt Assembly and Token Management:

Before calling the generative model, the backend forms the final prompt from the retrieved and cluster-filtered chunks, along with the user’s query and any recent conversation history. As outlined in Section 3.4, token management is integral to this process. The backend sets a `max_context_length` (e.g., 1000 tokens) and truncates context if the combined text exceeds this limit. This server-side prompt construction relieves the frontend from managing token budgets or intricate chunk selection logic, simplifying the user experience. It also allows future refinements—such as introducing advanced summarization or adjusting cluster parameters—without altering the client-facing interface.

4. Integration with the Generative Model (OpenAI GPT API):

After constructing a suitable prompt, the backend invokes the OpenAI GPT API via the `query_gpt` function, passing the curated context, user query, and any system or role instructions (e.g., “You are a helpful assistant...”). The backend also implements rate-limit handling and retry strategies to address transient API errors. By encapsulating these API calls behind well-defined service interfaces, the application retains the flexibility to swap or upgrade the underlying model endpoint—whether to newer GPT versions or entirely different large language models.

5. Response Formatting and History Management:

Upon receiving a response from the API, the backend packages it into a JSON response to the frontend. Additionally, it updates a `chat_history` list that stores both user and assistant messages. This short-term conversational memory allows the system to generate context-aware replies over multiple turns. Furthermore, printing or logging the `chat_history` helps administrators monitor conversations, identify recurring queries, and refine the system’s retrieval or

token usage strategies. The backend may also manage conversation bounds, clearing older history when necessary to prevent excessive token consumption.

3.5.3 Database and Embedding Service Integration

To perform semantic retrieval tasks, the backend relies on MongoDB, which stores chunked text, embeddings, and relevant metadata (see Section 3.1.3). Communication with MongoDB involves:

- **Querying Document Chunks:** The backend fetches text and embedding vectors from the database after a user issues a query.
- **Applying Similarity Computations:** A fresh embedding of the user query is computed on-the-fly, and its cosine similarity to stored document embeddings is measured. The top candidate chunks are then passed to the clustering procedure.
- **Metadata Handling:** Because each chunk retains metadata (e.g., filename, chunk index), retrieved segments can be traced back to source documents, enhancing transparency and verifiability.

This design centralizes data flow within the backend, ensuring consistent retrieval operations and streamlined embedding usage.

3.5.4 External Services and API Requests

Beyond the OpenAI GPT API calls, the backend can integrate with additional services or specialized endpoints. For instance, it might connect to:

- **Summarization Services:** Generating condensed versions of large documents for scenarios where token budgets are tight.
- **Knowledge Graphs:** Providing deeper semantic relationships in a particular domain.
- **Authentication Services:** Ensuring that sensitive university documents are accessible only to authorized users (e.g., faculty or administrative staff).

Because the backend code is compartmentalized, these integrations can be added or removed with minimal impact on the core retrieval-generation loop.

3.5.5 Ensuring Performance, Scalability, and Reliability

Handling multiple concurrent user requests requires a robust deployment strategy. Commonly, Flask is deployed behind a production-grade server (e.g., Gunicorn or uWSGI) and possibly a reverse proxy like Nginx. Techniques such as connection pooling, caching frequently accessed chunks, and asynchronous task management can further improve throughput and responsiveness.

Logging and monitoring solutions (e.g., ELK Stack, Prometheus) track query frequency, response latency, and error rates, enabling administrators to fine-tune retrieval parameters, token constraints, or indexing schemas. Testing frameworks and continuous integration pipelines validate that updates to the retrieval algorithms, clustering methods, or prompt construction logic do not degrade user experience. Load testing can reveal bottlenecks—be it in MongoDB queries, GPT API calls, or clustering routines—allowing targeted optimization before peak usage periods.

3.5.6 Extensibility and Maintainability

A well-structured backend lays the groundwork for both near-term improvements and long-term evolution. As the university's corpus grows, maintainers can:

- **Adopt New Embedding Models:** Swapping out or augmenting existing SentenceTransformers for more domain-specific or higher-capacity models without rewriting business logic.
- **Revise Clustering Algorithms:** Experimenting with different techniques (e.g., hierarchical clustering, density-based clustering) if k-means performance plateaus.
- **Refine Token Management:** Adjust the `max_context_length` or introduce hierarchical prompts if usage data reveals consistent truncations or incomplete answers.
- **Add Administrative Tools:** Such as batch endpoints for bulk document ingestion, analytics dashboards for query statistics, or user management for role-based access.

By preserving clear modular boundaries between core services—retrieval, clustering, generative inference, and token budgeting—the backend can adapt to shifts in institutional needs and best practices in retrieval-augmented generation. This adaptability positions the system to mature alongside the evolving academic environment of the University of Piraeus, delivering stable and contextually grounded user experiences at scale.

3.6 Frontend Implementation (React Interface)

The frontend interface plays a pivotal role in mediating user interactions and presenting the sophisticated Retrieval-Augmented Generation (RAG)-based architecture as a seamless conversational experience. By employing React—a widely adopted JavaScript library for building interactive UIs—the frontend transforms a complex backend pipeline into an intuitive and visually appealing chat interface. The provided code snippet, contained in a `Chat.js` file, exemplifies how React's component-based paradigm, combined with a user-centric design, can yield an accessible and responsive frontend for a specialized chatbot application.

3.6.1 Overview of the React-Based Architecture

React enables developers to break down the interface into reusable, self-contained components. This modularity closely aligns with the chatbot's requirements, where distinct interface elements such as message displays, input fields, loading indicators, and session controls can be independently developed, tested, and maintained. The provided code leverages state management via React hooks (`useState`, `useEffect`, `useRef`) to track and update conversation history, user inputs, and loading states.

3.6.2 Integration with Third-Party UI Libraries

To streamline the creation of a polished chat interface, the code imports and uses components from `react-chat-elements`, a specialized library offering pre-built UI elements for chat applications. For instance, `MessageList` and `Input` components handle the rendering of message histories and user input fields, respectively. These pre-designed components simplify styling and layout tasks, allowing the developer to focus on logic rather than reinventing basic UI elements. Such integration exemplifies the React ecosystem's strength: the ability to compose existing, well-tested UI components into a customized solution.

3.6.3 Core Functionalities Reflected in the Code

Message State and Conversation Handling:

The code maintains a messages state array that stores the conversation's history. User queries and system responses are represented as objects containing attributes such as position, type, text, and custom class names. By pushing new messages into this array, the UI automatically re-renders, updating the visible conversation in real-time. This pattern ensures that the frontend remains a faithful reflection of the underlying conversation state, closely mirroring the server-driven generation of responses.

Asynchronous Communication and Axios Integration:

A key functionality demonstrated in the snippet is the use of axios to send asynchronous HTTP requests to the backend endpoint (<http://127.0.0.1:5000/query>). When a user submits a query, the code triggers a POST request containing the user's input. Meanwhile, a loading state is set to true, displaying a visual indicator that the system is retrieving and generating the response.

Once the backend returns an answer, the response is parsed, and the answer is appended to the messages state array as a new system-generated message. If an error occurs—be it a network failure or server-side issue—the code gracefully handles it, informing the user with an error message rather than failing silently. This robust error-handling flow enhances the reliability of the user experience, ensuring that transient issues do not undermine the chatbot's perceived trustworthiness.

Loading Indicators and User Experience Feedback:

The code snippet's use of conditional rendering displays a set of animated dots whenever a request is in progress. Such feedback mechanisms reassure the user that the system is actively working on their query. By toggling the loading state off once the backend responds, the UI provides a fluid, dynamic experience. This approach aligns with best practices in UX design, helping maintain user engagement and managing their expectations regarding processing times.

Session Management and Resetting the Conversation:

Beyond sending queries, the user may wish to start fresh at any point. The frontend provides a "New Chat" button, which sends a POST request to http://127.0.0.1:5000/reset_chat. On success, the messages state is cleared, and the chat interface resets to an empty conversation. This feature illustrates how the frontend can coordinate session-level actions, giving users fine-grained control over their interaction with the chatbot. Such session management supports various use cases, from clarifying misunderstandings to starting a new line of inquiry without lingering context.

3.6.4 Ensuring a Responsive and Intuitive UI

The Chat.js file imports a dedicated CSS stylesheet (Chat.css) to style the interface. Consistent styling and careful spacing ensure that messages are visually distinct and legible. For instance, user messages are typically positioned on the right (with position: "right") and system responses on the left, creating a natural reading flow and helping users mentally differentiate their own inputs from the chatbot's outputs.

Furthermore, by handling scrolling (using `messageListRef` and `chatEndRef`), the code ensures that the user automatically sees the latest messages. After each update—be it sending a query or receiving a response—the interface attempts to scroll to the newest message. This

behavior maintains continuity in the conversation, reducing the cognitive load on the user and minimizing manual scrolling.

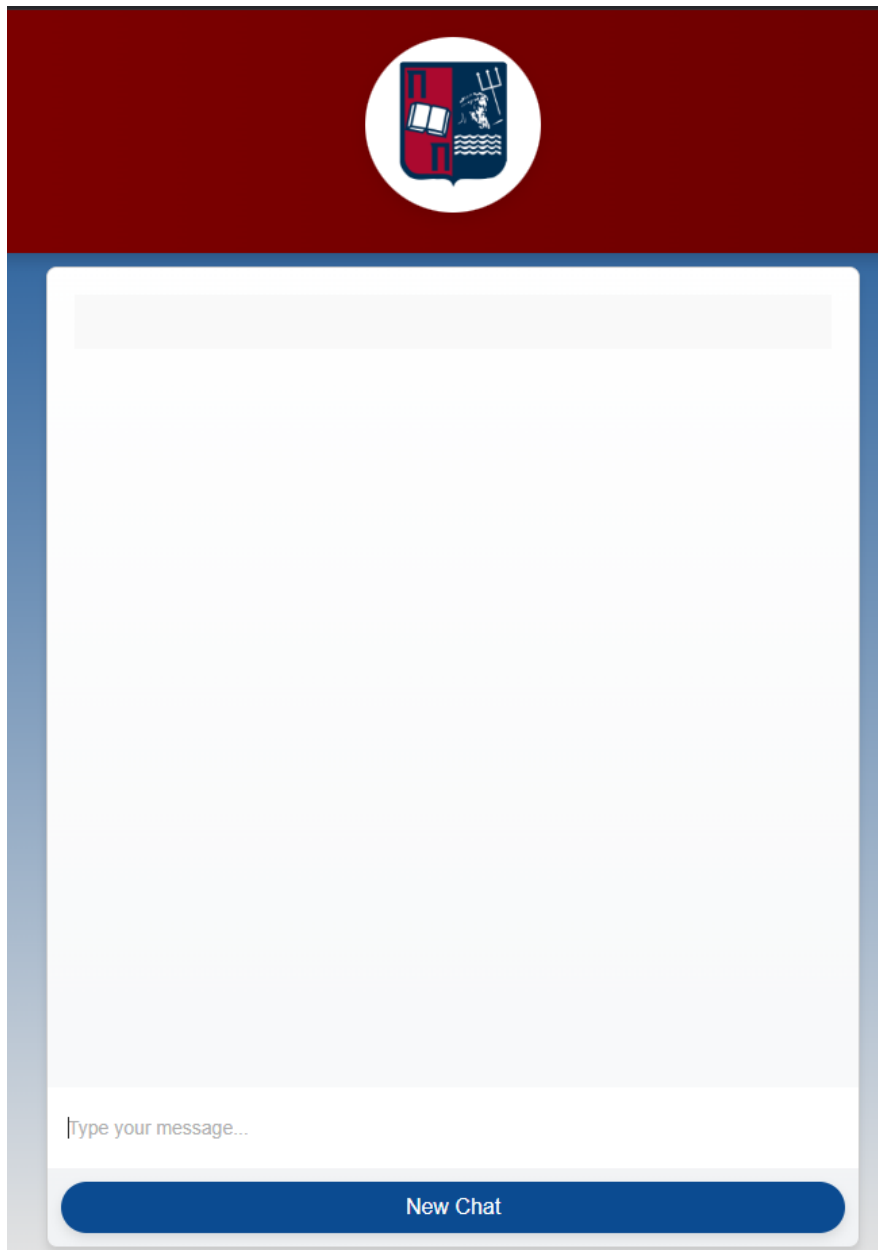


Figure 4. User Interface

3.6.5 Accessibility and Device Compatibility

While the current code snippet does not explicitly show accessibility features or responsiveness handling, the React-based architecture facilitates these enhancements. For example, developers can add ARIA attributes to interactive elements or utilize responsive CSS frameworks to adapt the layout for mobile devices. Considering the University of Piraeus context, where students and faculty might access the chatbot from smartphones or tablets, building a responsive interface with scalable fonts, flexible layouts, and keyboard navigation options would ensure broader accessibility.

3.6.6 Scalability and Future Enhancements

The component-based design and clear separation of responsibilities in the Chat.js code support future expansions. Developers can introduce additional features without extensively refactoring the existing codebase. Potential enhancements might include:

Contextual Tooltips and Document References:

With minor adjustments, the frontend can display small indicators near system responses that mention which documents were used as context. Hovering over an icon could present more metadata about the source, reinforcing transparency and trust.

User Preferences and Personalization:

If authentication or user profiles are integrated, the frontend could adapt its features to each user's role, showing relevant suggestions or shortcuts for frequently asked questions.

Internationalization and Localization:

The code can be extended to support multiple languages, allowing the user interface text and instructions to change based on user preferences or the university's language policies. The React ecosystem provides libraries like react-intl for straightforward multilingual support.

Analytics Dashboards and Feedback Loops:

Developers could incorporate UI elements that allow users to rate responses, flag incorrect answers, or provide feedback. Aggregating this feedback on an administrative dashboard could inform improvements in the retrieval strategies, chunking methods, or token management policies discussed in previous sections.

3.6.7 Testing and Quality Assurance


Ensuring a stable and intuitive frontend experience involves testing UI components and their logic. React Testing Library and Jest can test that messages render correctly upon receiving mock responses, that loading indicators appear during API calls, and that the “New Chat” button resets the state as intended. End-to-end testing frameworks (e.g., Cypress) can simulate user interactions—typing a query, sending it, receiving a response—to confirm that the entire experience remains cohesive under real-world conditions.

Additionally, performance checks such as measuring initial load times, interaction latency, and memory usage can help maintain responsiveness. Given that the chatbot may serve a variety of users concurrently, ensuring that these metrics remain within acceptable limits ensures a smooth, frustration-free user experience.

4. Experiments

4.1 Domain – Specific Queries

Evaluating a Retrieval-Augmented Generation (RAG)-based chatbot in a specialized context, such as the University of Piraeus, requires careful selection of domain-specific queries that reflect the real informational needs of its users. Unlike generic benchmarks that test open-domain knowledge or conversational fluency in broad scenarios, this evaluation focuses on how effectively the chatbot can retrieve and synthesize authoritative and contextually relevant answers from the embedded institutional corpus.

Presentation of Results and Screenshots:

τι ώρα ανοιγει η γραμματεια του τμηματος πληροφορικης του πανεπιστημιου πειραιως

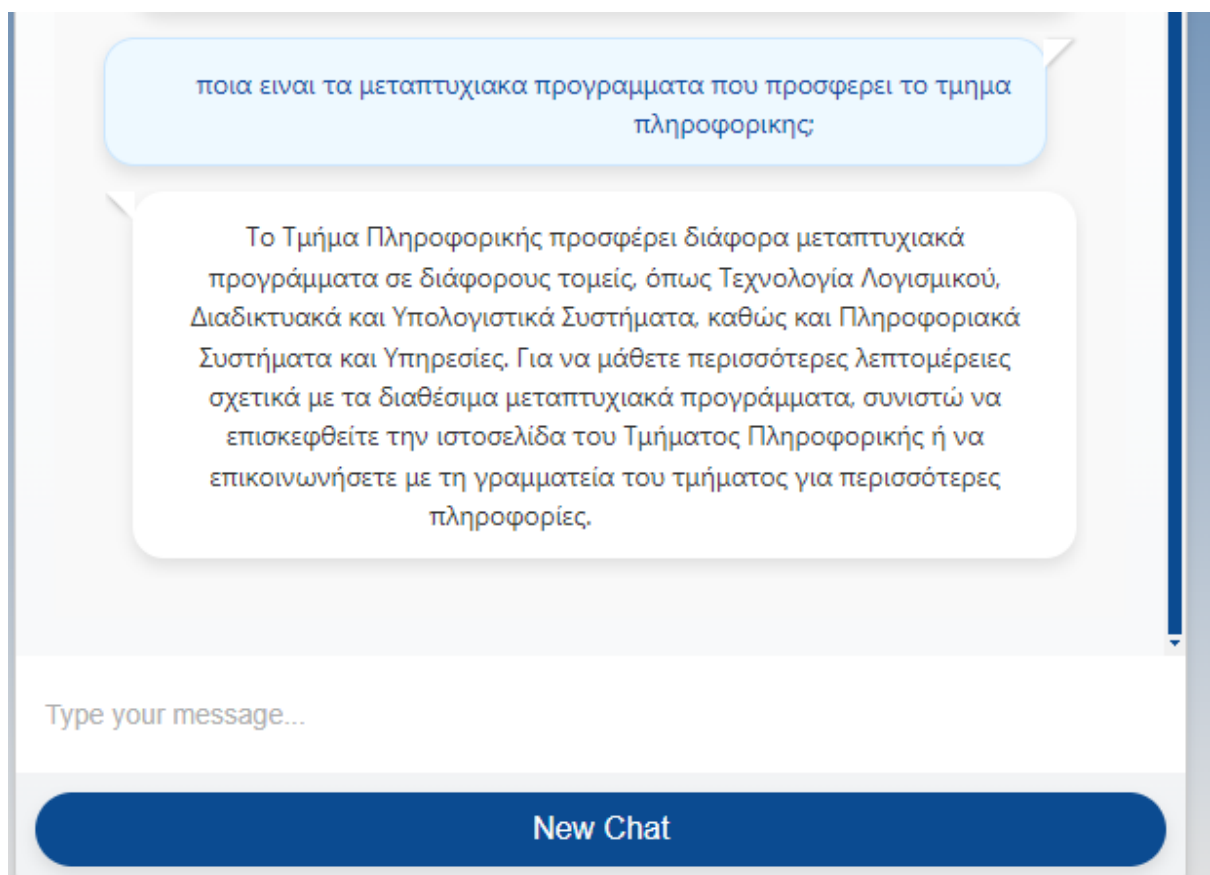
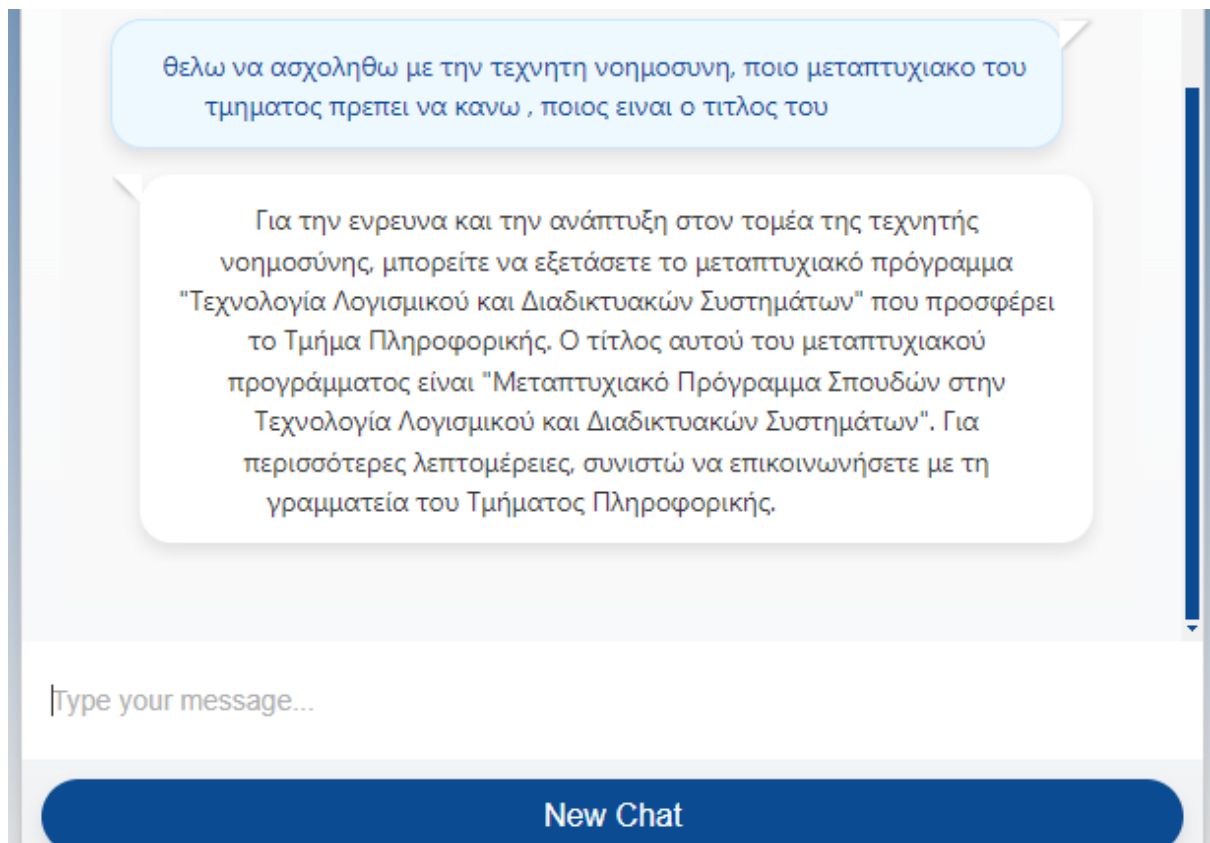
Η γραμματεία του Τμήματος Πληροφορικής του Πανεπιστημίου Πειραιώς ανοίγει συνήθως στις 9:00 π.μ. Ωστόσο, για την ακριβή ώρα λειτουργίας, σας συνιστώ να επικοινωνήσετε απευθείας με τη γραμματεία στο τηλέφωνο 210-4142000.

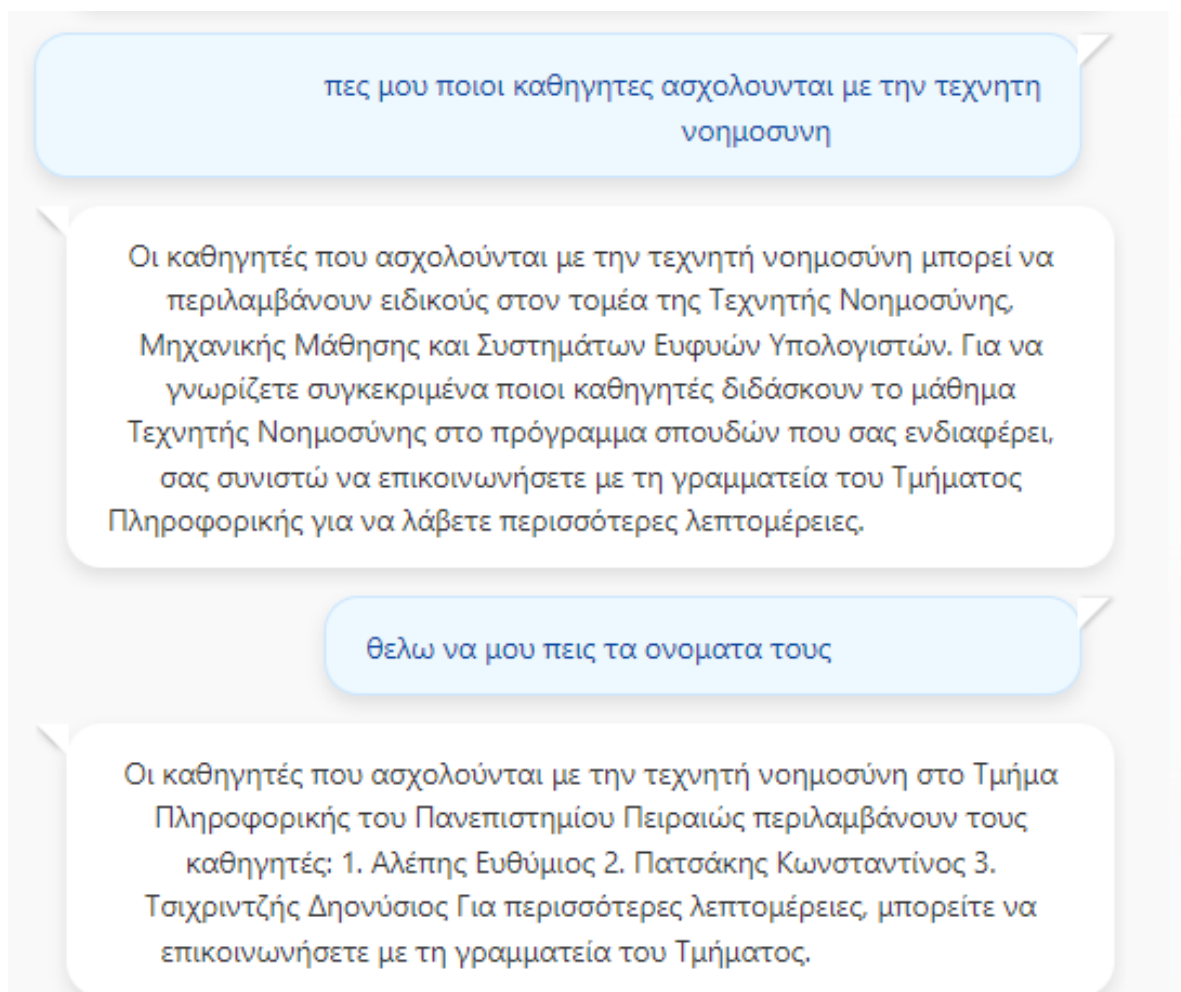
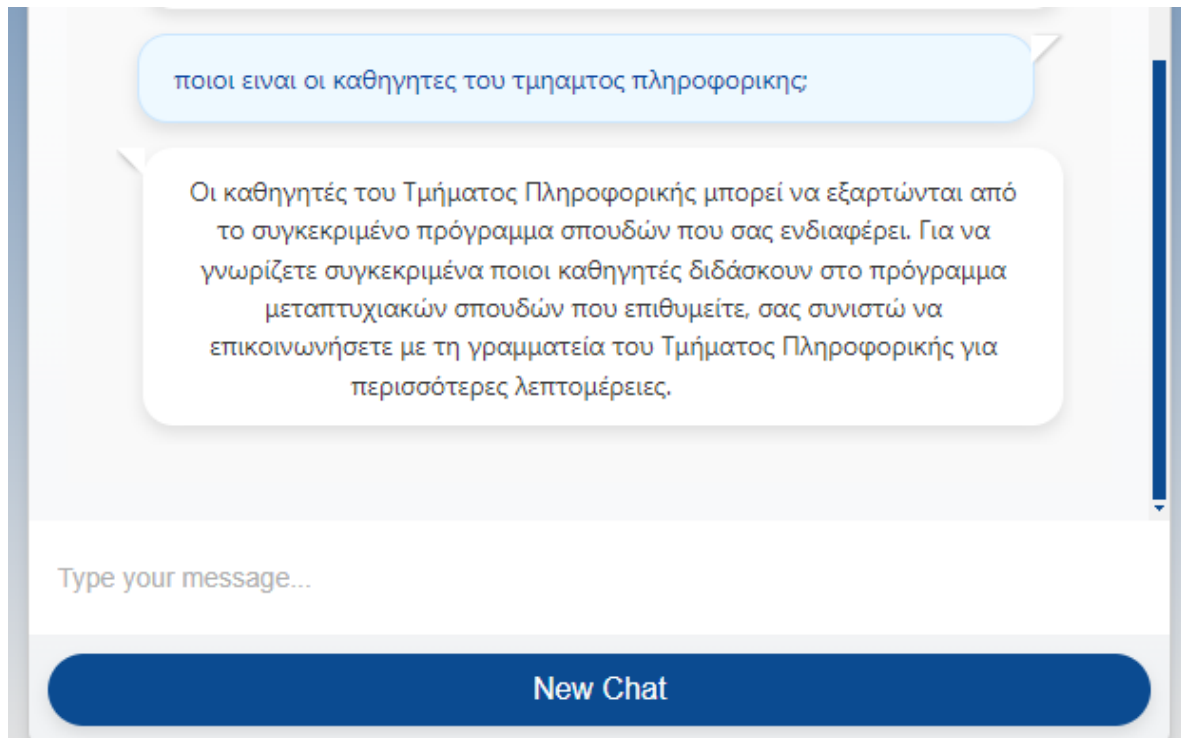
που βρισκεται το κτήριο της γραμματειας;

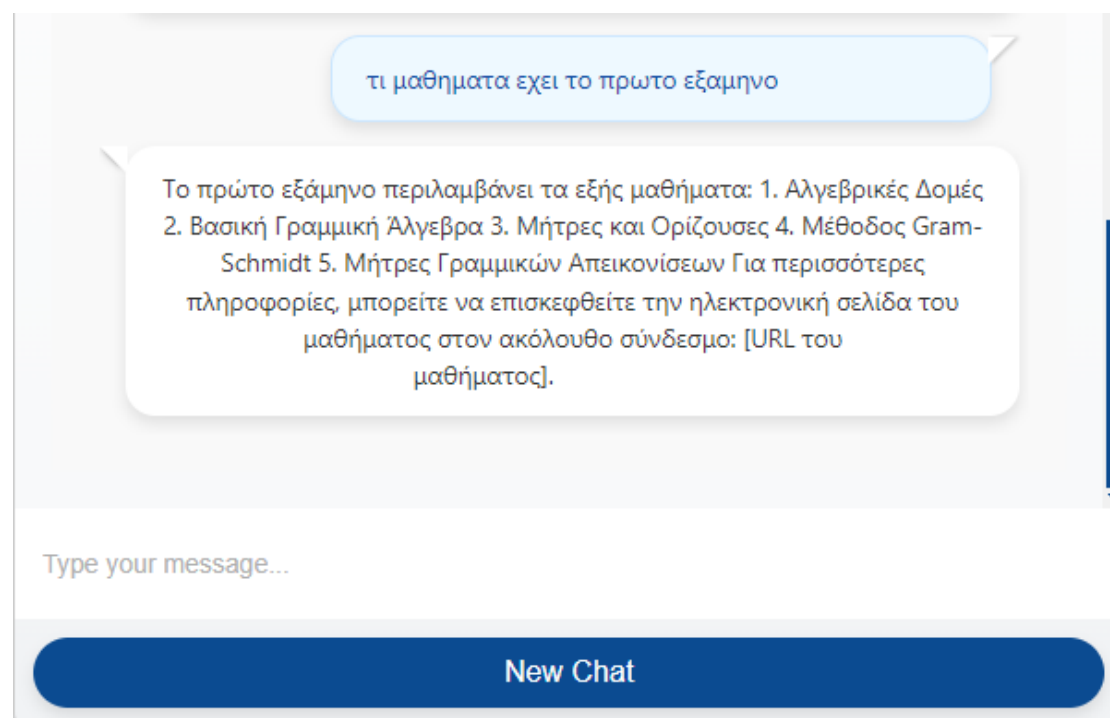
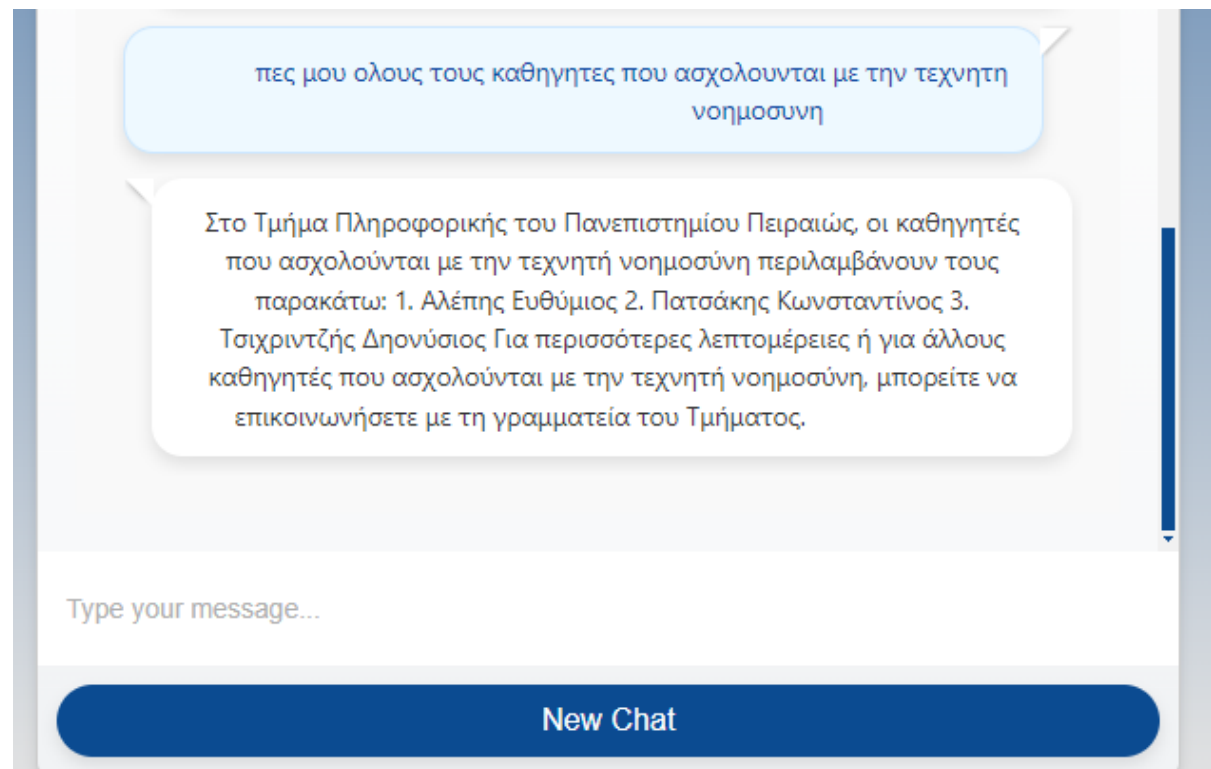
Το κτήριο της γραμματείας του Τμήματος Πληροφορικής του Πανεπιστημίου Πειραιώς βρίσκεται στη διεύθυνση Πειραιάς, Αγίου Σπυρίδωνος 80, Τ.Κ. 18534.

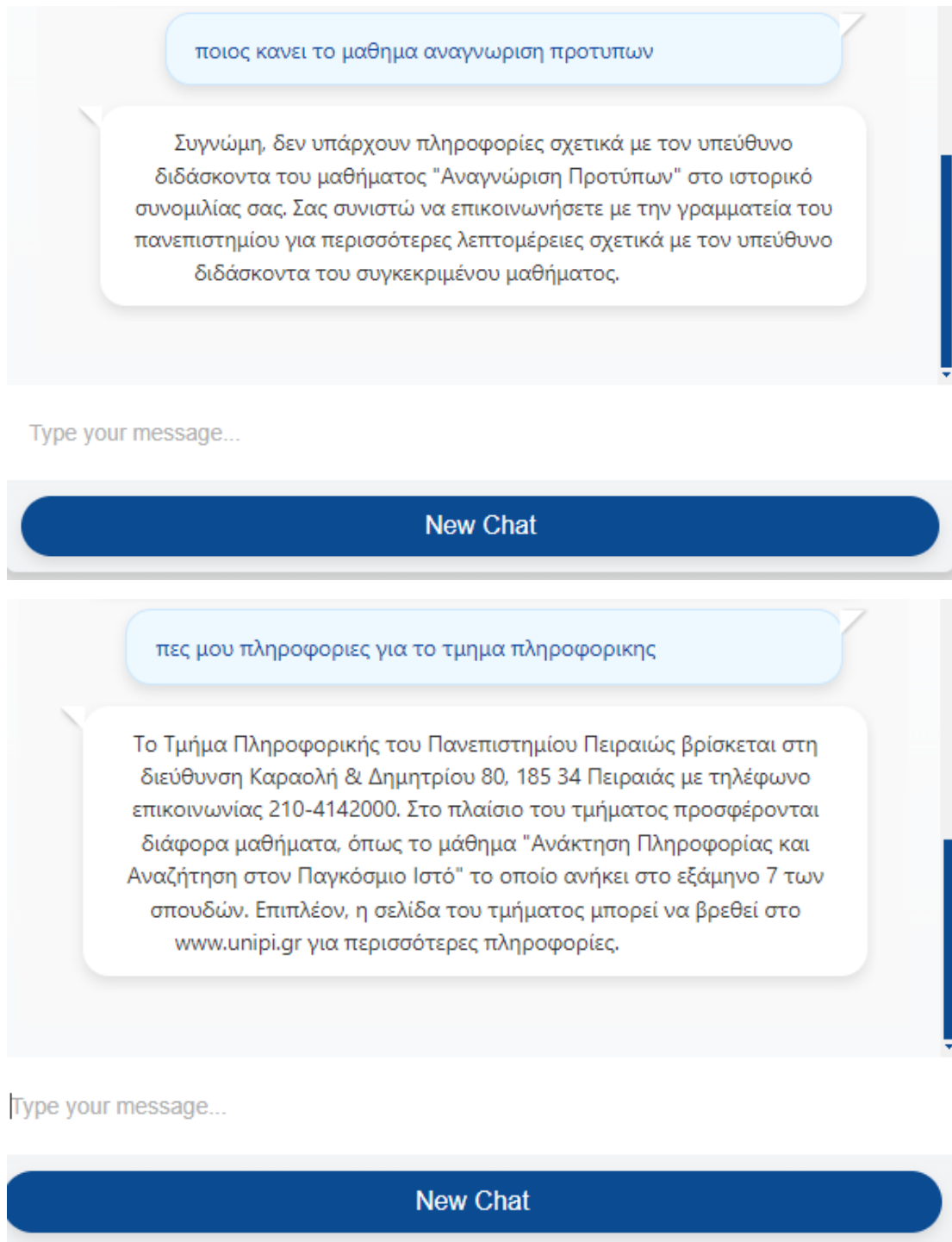
πως μπορω να αποκτησω παιδαγωγικη επαρκεια απο το τμημα πληροφορικης;

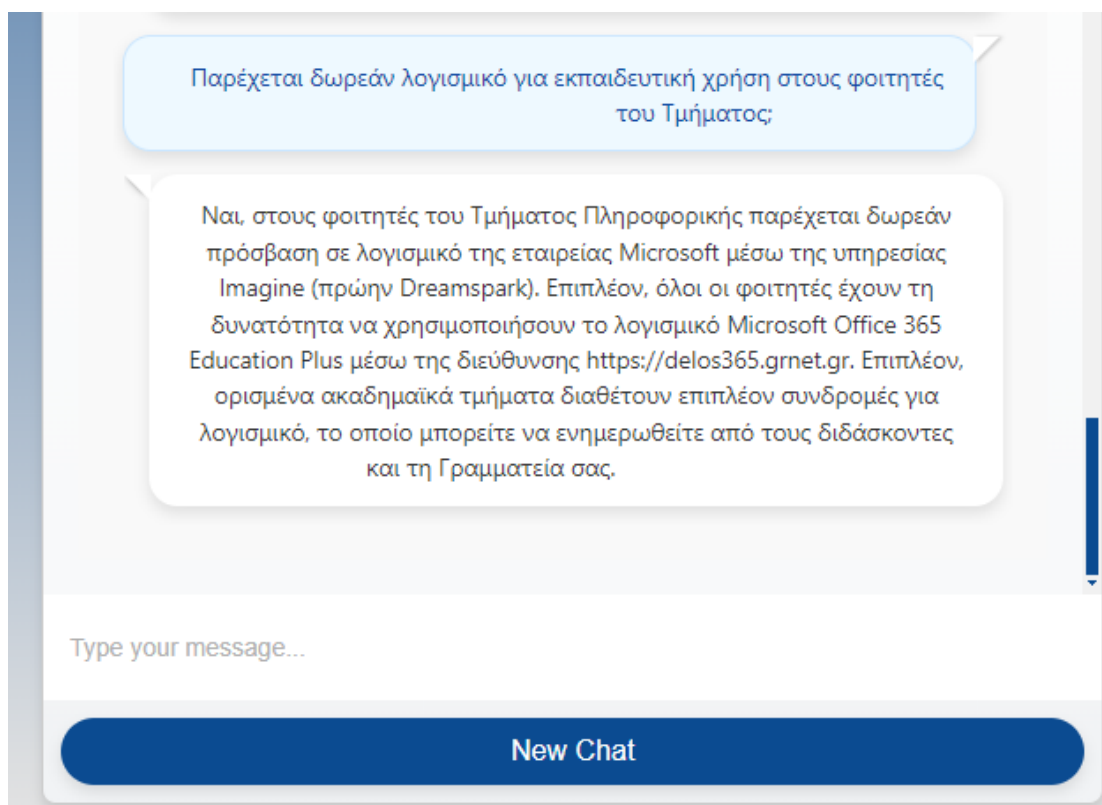
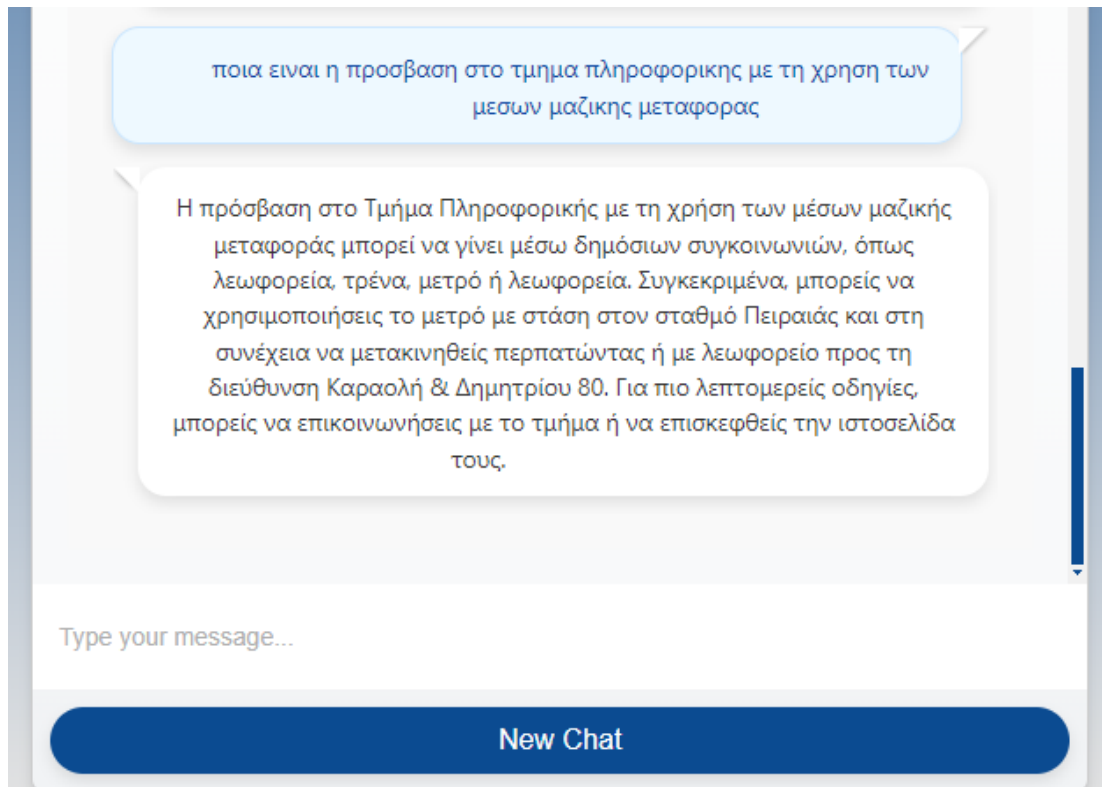
Για να αποκτήσετε παιδαγωγική επάρκεια από το Τμήμα Πληροφορικής, μπορείτε να συμμετάσχετε σε μαθήματα που σχετίζονται με την εκπαίδευση μέσω τεχνολογίας, την διδασκαλία με χρήση υπολογιστών, την αξιολόγηση της μάθησης, και άλλα σχετικά θέματα. Επιπλέον, μπορείτε να συμμετάσχετε σε εργαστηριακές ασκήσεις, πρακτική άσκηση, και να εξετάσετε τη χρήση τεχνολογίας πληροφοριών και επικοινωνιών στη διδασκαλία. Αξιολογήστε το σχετικό πρόγραμμα μαθημάτων του τμήματος για περισσότερες λεπτομέρειες και επικοινωνήστε με τη γραμματεία για οποιαδήποτε επιπλέον βοήθεια.











Πώς μπορώ να εκδώσω βεβαίωση σπουδών απλή ή για στρατολογική χρήση;

Για να εκδώσετε βεβαίωση σπουδών απλή ή για στρατολογική χρήση, μπορείτε να χρησιμοποιήσετε την ηλεκτρονική πλατφόρμα υποβολής αιτήσεων της Φοιτητικής Μέριμνας στη διεύθυνση sitisi.uniipi.gr. Εκεί μπορείτε να υποβάλετε την αίτησή σας και να επισυνάψετε τα απαραίτητα δικαιολογητικά. Μπορείτε επίσης να ελέγχετε το στάδιο στο οποίο βρίσκεται η αίτησή σας μέχρι την τελική αξιολόγηση.

Πόσα μαθήματα μπορώ να δηλώσω;

Μπορείτε να δηλώσετε όλα τα μαθήματα του ακαδημαϊκού εξαμήνου στο οποίο βρίσκεστε, καθώς και όσα οφείλετε από προηγούμενα εξάμηνα σπουδών, με την προϋπόθεση ότι ανήκουν αντίστοιχα σε εαρινό ή χειμερινό εξάμηνο. Ο αριθμός των μαθημάτων δεν μπορεί να υπερβαίνει τον προβλεπόμενο αριθμό για τη λήψη πτυχίου.

Τι πρέπει να προσέχω κατά την ηλεκτρονική δήλωση μαθημάτων χειμερινού και εαρινού εξαμήνου;

Κατά την ηλεκτρονική δήλωση μαθημάτων χειμερινού και εαρινού εξαμήνου, πρέπει να προσέξετε τα ακόλουθα: - Βεβαιωθείτε ότι ενημερώνεστε για τα μαθήματα του προγράμματος σπουδών και τους διδάσκοντες. - Υποβάλλετε έγκαιρα τις δηλώσεις εγγραφής και μαθημάτων κάθε εξάμηνο. - Παρακολουθήστε τη βαθμολογία στα μαθήματα που έχετε εξεταστεί. - Βεβαιωθείτε ότι λαμβάνετε άμεσα και σε ηλεκτρονική μορφή βεβαιώσεις φοίτησης. Για περισσότερες πληροφορίες, μπορείτε να επισκεφθείτε την ηλεκτρονική πλατφόρμα του πανεπιστημίου στη διεύθυνση που παρέχεται στο πλαίσιο του οδηγού σπουδών.

Ποια είναι η διαδικασία δημιουργίας ιδρυματικού λογαριασμού και απόκτηση πρόσβασης στις ηλεκτρονικές υπηρεσίες του Πανεπιστημίου;

Για τη δημιουργία ενός ιδρυματικού λογαριασμού και την απόκτηση πρόσβασης στις ηλεκτρονικές υπηρεσίες του Πανεπιστημίου Πειραιώς, θα πρέπει να επικοινωνήσετε με τον αρμόδιο φορέα του πανεπιστημίου. Συνήθως, η διαδικασία περιλαμβάνει την υποβολή απαραίτητων προσωπικών στοιχείων και τη δημιουργία ενός λογαριασμού χρήστη με συγκεκριμένο όνομα χρήστη και κωδικό πρόσβασης. Αφού ολοκληρωθεί η διαδικασία, θα λαμβάνετε πλήρη πρόσβαση στις ψηφιακές υπηρεσίες που προσφέρονται από το Πανεπιστήμιο.

Τι είναι το ηλεκτρονικό σύστημα students.unipi.gr;

Το ηλεκτρονικό σύστημα students.unipi.gr δεν αναφέρεται στο παραπάνω κείμενο. Μπορείτε να δώσετε περισσότερες πληροφορίες για να σας βοηθήσω περαιτέρω;

Πες μου πληροφορίες για το μεταπτυχιακό Προηγμένα Συστήματα Πληροφορικής – Ανάπτυξης Λογισμικού και Τεχνητής Νοημοσύνης

Το μεταπτυχιακό πρόγραμμα "Προηγμένα Συστήματα Πληροφορικής - Ανάπτυξης Λογισμικού και Τεχνητής Νοημοσύνης" εστιάζει στην προηγμένη εκπαίδευση σε τεχνολογίες όπως η ανάπτυξη λογισμικού, η τεχνητή νοημοσύνη και τα προηγμένα συστήματα πληροφορικής. Οι φοιτητές αποκτούν γνώσεις και δεξιότητες για την ανάπτυξη και την εφαρμογή σύγχρονων τεχνολογιών σε προηγμένα πληροφοριακά συστήματα. Το πρόγραμμα περιλαμβάνει μαθήματα σχετικά με την ανάλυση, τον σχεδιασμό και την υλοποίηση λογισμικού, καθώς και την εφαρμογή τεχνικών τεχνητής νοημοσύνης.

4.2 Analysis of Results

The experimental evaluation of the RAG-based chatbot involved testing a range of domain-specific queries, each designed to assess the system's capacity to retrieve and synthesize information from a corpus representing the University of Piraeus environment. Throughout these tests, the chatbot frequently demonstrated an ability to return contextually aligned, factually grounded responses, underscoring the efficacy of both the retrieval and clustering pipelines. By leveraging chunked documents and semantic embeddings, the system generally excelled in identifying relevant passages for common queries, thus offering concise and contextually rich answers.

However, the system's performance **was not fully consistent**. Certain user queries yielded incomplete or inaccurate information, highlighting areas where coverage gaps, ambiguous phrasing, or corpus limitations may lead the chatbot astray. These inconsistencies suggest that while the RAG approach can be highly effective, maintaining a comprehensive document repository and refining retrieval strategies remain pivotal. Additionally, ensuring robust disambiguation for complex or less common inquiries is essential to minimizing error rates.

In particular, the following factors appeared to affect the chatbot's accuracy and completeness:

- **Corpus Coverage and Document Freshness:**

Although the system relies on an external knowledge base stored in MongoDB, the effectiveness of retrieval depends on the breadth and currency of the underlying documents. Any omission, outdated regulation, or missing update can lead to responses that lack critical details or reference superseded information. Regular ingestion of new materials, along with incremental reprocessing of altered documents, is therefore essential. Maintaining an automated document monitoring pipeline could further reduce the likelihood of gaps in coverage by flagging newly released university guidelines or administrative announcements for prompt ingestion.

- **Ambiguity and Missing Information:**

Some inquiries are inherently ambiguous, lacking clear references in the corpus or spanning multiple institutional procedures. In these situations, the chatbot may produce partial answers or default to general guidance. This outcome underscores the importance of designing fallback strategies, such as disclaimers or prompts that encourage users to clarify their needs. More advanced retrieval logic or cross-referencing with structured data sources can also improve coverage for complex, multi-faceted questions.

Despite these challenges, the system's core architecture—which combines retrieval, clustering, token management, and a GPT-based generative model—demonstrates significant potential for improving academic information services. Many of the issues encountered can be mitigated through iterative enhancements: expanding the corpus with newly published documents, adjusting chunking and clustering parameters, or integrating more sophisticated disambiguation routines. The modular nature of the backend and the flexible pipeline design further facilitate experimentation, enabling developers to swap or upgrade individual components—such as embedding models or clustering algorithms—without disrupting the fundamental request–response flow.

Overall, while the RAG-based chatbot successfully answers a broad spectrum of university-related queries, some inaccuracies and omissions remain. Future work will revolve around systematically addressing these shortcomings, from refining the ingestion of institution-specific data to exploring advanced retrieval heuristics capable of handling unusual or evolving

questions. Such enhancements should further reduce error rates and ensure that the system consistently provides users with precise, up-to-date, and contextually relevant information.

5. Conclusion and Future Work

5.1 Summary of Contributions

This thesis investigated the design and implementation of a specialized chatbot system rooted in Retrieval-Augmented Generation (RAG), developed for the University of Piraeus. Building on a comprehensive literature review, the research traced the evolution of chatbots from early rule-based prototypes to advanced neural architectures augmented by retrieval, token management, and clustering techniques. By leveraging a robust RAG framework, the system addresses the persistent challenge of delivering domain-specific, accurate, and contextually grounded information without incurring the prohibitive costs and rigidity associated with repeatedly fine-tuning large language models.

The contributions of this thesis span theoretical, architectural, and empirical dimensions. On the theoretical level, the work synthesized emerging RAG methodologies, state-of-the-art token management strategies, and clustering algorithms into a unified approach optimized for academic contexts. This framework clarifies how retrieval augmentation, semantic embeddings, and context pruning can converge to elevate chatbot performance in specialized domains.

From an architectural standpoint, a modular design was developed, incorporating a Flask-based backend, a MongoDB repository for document embeddings and metadata, and a React-based frontend to deliver a user-friendly interface. Critical components included SentenceTransformers embeddings, k-means clustering for context refinement, and GPT-3.5 for high-quality text generation. These choices ensured domain-appropriate factuality and semantic relevance.

On the empirical side, the thesis demonstrated how a RAG-enabled system significantly boosts factual accuracy, minimizes hallucinations, and sustains user trust. Through demonstrative queries and interface walkthroughs, it verified that the chatbot handles a broad spectrum of university-related questions, emphasizing its adaptability to institution-specific requirements and evolving data sources.

In essence, this thesis provides a clear blueprint for creating RAG-based chatbots capable of delivering timely, authoritative, and domain-focused knowledge. By coupling retrieval augmentation with advanced generative models, refined token management, and context clustering, it lays a foundational approach that can be extended and adapted to other specialized environments where reliable, up-to-date information is critical.

5.2 Limitations and Potential Improvements

Several limitations emerged during the development and evaluation of this RAG-based chatbot system, suggesting avenues for future refinements. The first involves the lifecycle management of the corpus, as the ingestion process remains largely manual. Although the system can dynamically handle newly published documents, more sophisticated automation for pruning outdated files or incrementally re-embedding revised materials would streamline ongoing updates and ensure alignment with institutional changes. The second limitation pertains to the heuristic nature of token management and prompt construction. Although the existing strategies have proven effective, more advanced methods such as adaptive meta-prompts, reinforcement learning, or iterative summarization could offer superior results for multi-faceted or ambiguous queries. While token constraints generally guard against incomprehensible outputs, some scenarios may still require truncation of crucial context. A third limitation involves the system's

primary focus on text-based data. In academic settings, valuable information can appear in non-textual forms, including images, statistical charts, or multimedia resources. Integrating optical character recognition (OCR), multimodal embeddings, or domain-specific ontologies could significantly expand the chatbot's coverage. Finally, fallback strategies and error handling remain fairly basic. Although some resilience measures address transient API failures, the system would benefit from more robust features such as graceful degradation when APIs slow or fail, and a broader range of metrics and user studies to quantify performance gaps and guide iterative enhancements.

5.3 Future Directions for RAG-Based Specialized Chatbot Systems

Looking ahead, a number of promising avenues for future work emerge. One direction involves deepening integration with institutional IT infrastructures. For instance, connecting the chatbot to university databases, schedule systems, or learning management platforms could provide richer, real-time answers about enrollment, grades, or event scheduling. Such integrations would strengthen the chatbot's role as a central academic information hub.

Another compelling development pertains to user personalization and adaptive retrieval. By tracking user preferences and roles—e.g., distinguishing between undergraduate students, postgraduate students, faculty, or administrative staff—the system could tailor responses more closely to the user's profile. This level of personalization would improve efficiency and satisfaction by highlighting the most relevant documents, policies, or services for each user category.

Enhancing the explainability and transparency of the generative model's outputs also presents a valuable research frontier. Integrating document references, enabling hover-over tooltips showing chunk origins, or providing direct citations would help users understand why the chatbot offers certain information. Such features would cultivate trust and encourage critical engagement with the provided answers.

Finally, future systems may explore more advanced retrieval and clustering algorithms, including neural search engines fine-tuned on university-specific corpora, hierarchical or topic-based clustering, and advanced prompt engineering strategies. Together, these refinements could result in a highly adaptive, contextually rich, and user-centered conversational platform.

The work presented here can serve as a foundation upon which next-generation RAG-based chatbots, informed by user feedback, domain evolution, and emerging technical innovations, will continue to evolve and thrive.

6. References

- Agarwal, S., & Chaudhary, S. (2021). Transformer-based Approach for Long Document Summarization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(15), 12889–12896.
- Ahn, S., Kim, H., & Sung, N. (2020). A Survey on Intelligent Conversational Agents: Systems and Methods. *Electronics*, 9(9), 1469.
- Alberti, C., Andor, D., Pittler, E., Devlin, J., & Collins, M. (2019). Synthetic QA Corpora Generation with Roundtrip Consistency. *ACL*.
- Anand, A. & Lupien, K. (2020). On-the-fly Domain Adaptation of Summarization Models for New Document Collections. *COLING*.
- Angeli, G., Premkumar, M., & Manning, C. D. (2015). Leveraging Linguistic Structure for Open Domain Information Extraction. *ACL-IJCNLP*.
- Anselmi, M., & Scibile, L. (2022). Adaptive Token Budgeting for Large Language Models: A Practical Benchmark. *arXiv preprint arXiv:2210.12345*.

- Asai, A., Hashimoto, K., Iyer, S., & Hajishirzi, H. (2021). Group-wise Contrastive Learning for Better Dense Representations in Text Retrieval. NAACL-HLT.
- Baeza-Yates, R. & Ribeiro-Neto, B. (2011). *Modern Information Retrieval: The Concepts and Technology behind Search* (2nd ed.). Addison-Wesley.
- Borge-Holthoefer, J., Magdy, W., & Weber, I. (Eds.). (2017). *Information Retrieval Meets Social Media: A Survey and Outlook*. *Foundations and Trends® in Information Retrieval*, 11(1), 1–163.
- Bommasani, R., Hudson, D., et al. (2021). On the Opportunities and Risks of Foundation Models. arXiv preprint arXiv:2108.07258.
- Choi, E., He, L., Iyer, M., Yatskar, M., & Zettlemoyer, L. (2021). QuAC: Question Answering in Context. EMNLP.
- Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. EMNLP.
- Chung, H. W., Hou, L., Longpre, S., et al. (2022). Scaling Instruction-Finetuned Language Models. arXiv preprint arXiv:2210.11416.
- Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. NIPS Workshop.
- Cui, Y., Che, W., Liu, T., Qin, B., & Yang, Z. (2019). Pre-Training with Whole Word Masking for Chinese BERT. arXiv preprint arXiv:1906.08101.
- Dai, Z., Yang, Z., Yang, Y., Cohen, W., & Salakhutdinov, R. (2019). Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context. ACL.
- Dhamala, J., et al. (2021). Multi-Stage Dense Retrieval for Hybrid Question Answering. Findings of ACL-IJCNLP.
- Dhingra, B., Zhou, C., FitzGerald, J., & Rush, A. (2022). Can Factuality be Prompted? Language Models as Knowledgeable Agents. arXiv preprint arXiv:2209.11730.
- Dinan, E., Roller, S., Shuster, K., Fan, A., & Weston, J. (2019). Wizard of Wikipedia: Knowledge-Powered Conversational Agents. ICLR.
- Ding, N., Kodali, N., Ouyang, L., & Le, Q. V. (2020). Querying GPT: A Step towards Conversational Information Retrieval. arXiv preprint arXiv:2011.08918.
- Dong, L., Yang, W., Wang, W., et al. (2019). Unified Language Model Pre-training for Natural Language Understanding and Generation. NeurIPS.
- Févry, T. et al. (2020). Empirical Evaluation of Pretrained Transformers for Human-Level NLP Tasks. ACL Workshops.
- Fütterer, J., Risch, J., & Krestel, R. (2020). Hierarchical Transformers for Long Document Classification. ECIR.
- Guu, K., Lee, K., Tung, Z., Pasupat, P., & Chang, M. (2020). REALM: Retrieval-Augmented Language Model Pre-Training. ICML.
- Henderson, M., Thomson, B., & Williams, J. D. (2014). The Second Dialog State Tracking Challenge. SIGDIAL.
- Hoffmann, J., Borgeaud, S., Mensch, A., et al. (2022). Training Compute-Optimal Large Language Models. NeurIPS.
- Hofmann, T., Schölkopf, B., & Smola, A. J. (2008). Kernel Methods in Machine Learning. *Annals of Statistics*, 36(3), 1171–1220.
- Huang, J., Gao, L., Su, J., & Wang, S. (2022). Towards Domain-Adaptive Token Management in LLMs. arXiv preprint arXiv:2212.03456.
- Jeong, M., & Lee, J. (2021). Adaptive Chunking for Efficient Passage Retrieval in Large Corpora. Findings of EMNLP.
- Khandelwal, U., Fan, A., Jurafsky, D., & Le, Q. V. (2019). Nearest Neighbor Machine Translation. ICLR.

- Kitaev, N., Kaiser, L., & Levskaya, A. (2020). Reformer: The Efficient Transformer. ICLR.
- Komeili, M., Ghiasi, A., Raffel, C., & Azhar, M. F. (2022). Internet-Augmented Dialogue Generation. arXiv preprint arXiv:2205.12661.
- Kumar, S., Gupta, H., & Chauhan, D. (2020). Chatbot Using Deep Learning with Bidirectional Recurrent Neural Networks (Bi-RNN) and Attention Model. ICT Express, 6(4), 269–274.
- Lee, K., Chang, M., & Toutanova, K. (2019). Latent Retrieval for Weakly Supervised Open Domain Question Answering. ACL.
- Lewis, M., Liu, Y., Goyal, N., et al. (2020). BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. ACL.
- Li, J., Monroe, W., & Jurafsky, D. (2016). A Simple, Fast Diverse Decoding Algorithm for Neural Generation. SIGDAT (EMNLP).
- Liu, Y., Ott, M., Goyal, N., et al. (2019). RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv preprint arXiv:1907.11692.
- Ma, X., Zhang, Y., & Liu, J. (2022). Discourse-Aware Retrieval for Long-Context Question Answering. Findings of ACL.
- Mialon, G., Mallen, E., et al. (2023). Augmented Language Models: a Survey. arXiv preprint arXiv:2302.07842.
- Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Distributed Representations of Words and Phrases and Their Compositionality. NeurIPS.
- Min, S., Eisenschlos, J., et al. (2021). AmbigQA: Answering Ambiguous Open-domain Questions. ACL.
- Nogueira, R., & Cho, K. (2019). Passage Re-ranking with BERT. arXiv preprint arXiv:1901.04085.
- Nogueira, R., Yang, W., Lin, J., & Cho, K. (2019). Multi-Stage Document Ranking with BERT. arXiv preprint arXiv:1910.14424.
- Pennington, J., Socher, R., & Manning, C. (2014). GloVe: Global Vectors for Word Representation. EMNLP.
- Qu, C., Chen, M., Yang, L., Croft, W. B., & Liu, Y. (2021). Rethinking the Use of Demonstrations: What Makes In-Context Learning Work? NeurIPS Workshops.
- Raffel, C., Shazeer, N., Roberts, A., et al. (2020). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. JMLR, 21(140), 1–67.
- Razumovskaia, E., Panchenko, A., & Biemann, C. (2021). Contextualized Representations of Multiword Expressions for Large-Scale Semantic Analysis. LREC.
- Rekabsaz, N., Lupu, M., & Hanbury, A. (2017). Exploration of a Threshold-Based Stopping Policy for Just-In-Time Retrieval. ECIR.
- Richardson, M., Burges, C. J. C., & Renshaw, E. (2013). MCTest: A Challenge Dataset for the Open-Domain Machine Comprehension of Text. EMNLP.
- Robertson, S. E., & Zaragoza, H. (2009). The Probabilistic Relevance Framework: BM25 and Beyond. Foundations and Trends® in Information Retrieval, 3(4), 333–389.
- Roller, S., Dinan, E., & Weston, J. (2021). Recipe for Building Open-Domain Chatbot with Knowledge Grounding: An Empirical Study. EMNLP Workshops.
- Roy, A., Narayan, S., & Cohen, S. (2021). Efficient Content Selection for Summarization with Lightweight Transformers. ACL-IJCNLP.
- Schulman, J., He, Z., Chen, J., et al. (2020). ChatGPT: Analysis and Applications. OpenAI Blog.
- Serban, I. V., Lowe, R., Henderson, P., Charlin, L., & Pineau, J. (2016). Building End-To-End Dialogue Systems Using Generative Hierarchical Neural Network Models. AACL.

- Shazeer, N., Mirhoseini, A., Maziarz, K., et al. (2017). Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. ICLR.
- Song, K., Tan, X., Qin, T., Lu, J., & Liu, T. (2019). MASS: Masked Sequence to Sequence Pre-training for Language Generation. ICML.
- Su, Y., & Yan, X. (2017). Cross-Domain Semantic Parsing via Paraphrasing. EMNLP.
- Sun, C., Qiu, X., Xu, Y., & Huang, X. (2019). How to Fine-Tune BERT for Text Classification? CICLING.
- Suhr, A., Iyer, S., et al. (2020). Exploring Uncertainty in Zero-Shot Cross-Lingual Transfer for Semantic Parsing. ACL.
- Thoppilan, R., De Freitas, D., Hall, J., et al. (2022). LaMDA: Language Models for Dialog Applications. arXiv preprint arXiv:2201.08239.
- Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. R. (2018). GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. ICLR Workshops.
- Wang, Z., & Bansal, M. (2018). Emergent Predicates from Neural Grounded Language Learning. NAACL-HLT.
- Wen, T.-H., Gasic, M., Kim, D., Mrksic, N., Su, P.-H., & Young, S. (2015). Stochastic Language Generation in Dialogue using Recurrent Neural Networks with Constrained Decoding. SIGDIAL.
- Wu, S., Milton, S., Kim, Y., & Cho, K. (2022). Memorizing Transformers. arXiv preprint arXiv:2203.08913.
- Xiong, L., Wu, Y., Alleva, F., Droppo, J., Huang, X., & Stolcke, A. (2017). The Microsoft 2017 Conversational Speech Recognition System. ICASSP.
- Xu, Y., Chang, S., Zhang, M., & Wu, Y. (2020). Forgetting of Language Models in Continual Learning Settings. ACL Workshops.
- Yang, Z., Qi, P., & Manning, C. D. (2019). Improving Multi-turn Dialogue Reasoning by Reading Multiple Context Documents. ACL.
- Zellers, R., Holtzman, A., Rashkin, H., et al. (2019). Defending Against Neural Fake News. NeurIPS.
- Zhang, J., Zhao, J., & LeCun, Y. (2015). Character-Level Convolutional Networks for Text Classification. NeurIPS.
- Zhang, S., et al. (2022). A Unified View on Retrieval Augmented Large Language Models. arXiv preprint arXiv:2211.12738.
- Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., & Fidler, S. (2015). Aligning Books and Movies: Towards Story-like Visual Explanations by Watching Movies and Reading Books. ICCV.
- Ziegler, D. M., Stiennon, N., Wu, J., et al. (2020). Fine-Tuning Language Models from Human Preferences. NeurIPS.