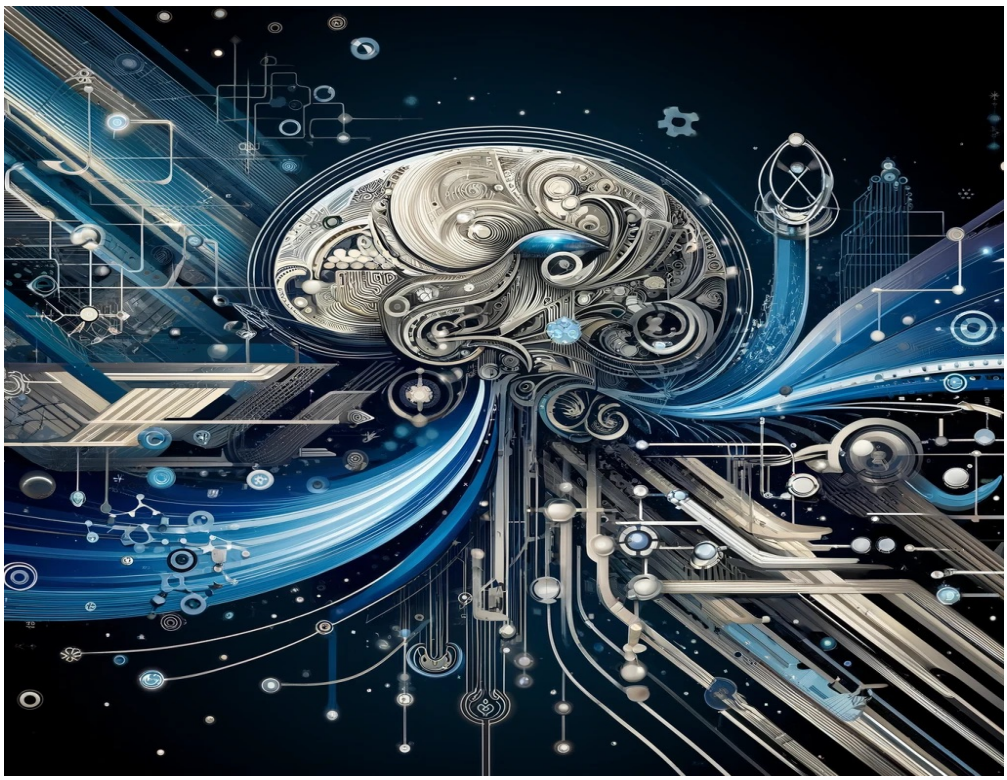

Exploring Patterns in LLM Integration

A study on architectural considerations and design patterns in LLM dependent applications

Master's thesis in Computer Science

Sundarakrishnan Ganesh

Robert Sahlqvist

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024
www.chalmers.se

Master's thesis 2024

LLM Integration: Unveiling Architectural Patterns and Design Considerations

Exploring LLM Integration Architectures and Design Considerations to Foster Robust and Scalable Applications

Sundarakrishnan Ganesh, Robert Sahlqvist

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer science and technology
Chalmers University of Technology & University of Gothenburg
Gothenburg, Sweden 2024

LLM Integration: Unveiling Architectural Patterns and Design Considerations
Exploring LLM Integration Architectures and Design Considerations to Foster Robust and Scalable Applications
Sundarakrishnan Ganesh Robert Sahlqvist

Supervisor: Robert Feldt
Examiner: Hans-Martin Heyn

Master's Thesis 2024
Department of Computer science and technology
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An image symbolizing the central intelligence of LLMs, the interconnected systems, and the innovative design patterns crucial for scalable and robust application development.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2024

LLM Integration: Unveiling Architectural Patterns and Design Considerations
Exploring LLM Integration Architectures and Design Considerations to Foster Robust and Scalable Applications
Sundarakrishnan Ganesh
Robert Sahlqvist
Department of Computer science and Technology
Chalmers University of Technology

Abstract

The surge in the integration of Large Language Models (LLMs) into software applications marks an evolution in the software development industry, leading to research into architectural frameworks and design patterns suitable for these technologies. This study aims to analyze the implementation of LLMs in software applications, exploring the varying architectures and design patterns that enhance the quality of these integrations. Despite the widespread adoption of LLMs, as highlighted by rapid user growth and diverse applications ranging from chatbots to complex problem-solving tools, a gap remains in the systematic exploration of architectural strategies. This research aims to bridge this gap by analyzing existing LLM applications, identifying architectural patterns and traces of cognitive architectures, and examining how these can be adapted for application development. Through a combination of literature review and structural analysis, this study seeks to offer insights into the architectural underpinnings that support successful LLM integration, thereby contributing to the broader discourse on software architecture in the age of advanced artificial intelligence technologies. With this we provide a library of design patterns tailored to different use-cases of LLM integration in applications.

Keywords: Large Language Models (LLMs), Generative AI (GenAI), Software Architecture, Design Patterns, LLM Integration, Cognitive Architectures, Artificial Intelligence (AI), Multi-Agent Systems, Prompt Engineering, In-Context Learning, Retrieval-Augmented Generation (RAG), Transformer Models, GPT (Generative Pre-trained Transformer), Open Source Projects, Scalability, Maintainability, Software Development, API (Application Programming Interface), Empirical Analysis, Case Study, Literature Review, LLM Applications, MetaGPT, DroidAgent, AutoGen, InvokeAI, Cognitive Architecture Traces, Architectural Frameworks, Industry Applications, OpenAI API

Acknowledgements

We extend our sincere gratitude to everyone who made this thesis possible. We especially thank our supervisor, Dr. Robert Feldt, for his invaluable contributions, stimulating suggestions, and encouragement. His expertise in design thinking and technology significantly guided our research. Moreover, I (Sundarakrishnan) would also like to thank my beloved parents. My father who has been exceptionally supportive and helpful with my research as he is in the same field and my mother for her wise counsel during this time. Furthermore, I would also use this platform to thank my roommates, my closest friends and my thesis partner, who were very understanding and helped me through this journey. I (Robert) would like to thank my friends and family for being there and helping me when they can. And I would like to thank my thesis partner as well for making this thesis possible.

Sundarakrishnan Ganesh, Gothenburg, June 2024 & Robert Sahlqvist, Gothenburg, June 2024

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

LLM	Large Language Model
AI	Artificial Intelligence
ML	Machine Learning
CNN	Convolutional Neural Networks
RNN	Recurrent Neural Networks
GenAI	Generative AI
GPT	Generative Pre-trained Transformer
API	Application Programming Interface
AGI	Artificial General Intelligence
RAG	Retrieval Augmented Generation

Contents

List of Figures

List of Tables

1

Introduction

1.1 Introduction

Since early 2023, the use of LLMs and GenAI has significantly increased in the industry. The release of OpenAI's GPT-3 model marked a turning point, leading to a rapid growth in LLM adoption. According to Marr, ChatGPT reached 100 million users within two months of its release, highlighting the strong interest in and potential of these technologies [1].

On the other hand, due to the rise in complex softwares today, it is ideal for companies to adopt certain architectures and design patterns to improve the overall maintainability and scalability of their applications. In this document, these kinds of applications are what we classify as LLM-based applications. To provide a better idea of what these applications could do, we can consider a few examples. Consider a chatbot made employing the OpenAI API. This would be an LLM-based application as the responses generated by the chatbot are not preprogrammed, rather it uses an LLM to generate its responses. This would help to provide more realistic responses to the user command. Moreover, LLM integration in an application need not be connected to an API, rather also make use of their own custom LLM model. So an LLM based application is essentially any software product which utilises LLMs to some degree, either as their core component to provide services or involving a minor use to provide solutions for specific use-cases.

Such applications can be enhanced and scaled using several existing architectures / design patterns. Since the surge of LLMs, more and more companies are resorting to LLM solutions for their problem statements, according to an article from Welocalize [2]. This growth of interest towards LLMs is unsurprising considering how they could be efficient and would essentially require the right prompt for it to present a viable solution. This inturn gave birth to another subdomain of computer science which deals with prompt engineering, which enables a user to "talk" to an LLM in the right way to obtain optimal results. According to Bertalan Meskó, prompt engineering is a crucial skill and could greatly improve the medical field [3]. This is advantageous since people can actually communicate to a computer in plain English rather than going through APIs or different protocols to get a result. With this we can argue how LLMs can aid in software development or even in general industry workflows. Consider MetaGPT, an application built using OpenAI's models and helps in generating user stories, requirements, competitive analysis etc, from a single prompt message [4].

However, most of the literature centred around the utilisation of LLMs focus mostly

on the application and usefulness of LLMs. Very little light has been shed on ideal architectures and design patterns which can be used while building LLM based applications, since it is a very new topic of discussion. Hence, the goal of this study is to focus on studying current implementations of LLM integration in order to make attempts at extracting useful patterns, or possibly observe the adaptation of established patterns. The next section describes the problem statement, while building up to the actual purpose of this study.

1.2 Problem

Software architecture is a crucial building block in a project, which would lay the foundation of the entire project. It is important to recognize, as noted by Frederick Brooks [5], that no universal 'silver bullet' architecture exists that is suitable for all use-cases. This claim is basically supported by the diverse objectives and requirements characteristic of LLM-based applications. These applications require tailored architectural approaches to address specific challenges and goals effectively. Here effectiveness is a measure of how well the architectures help the applications perform their functions and meet the desired objectives with accuracy and relevance to the problems they are designed to solve. For example, the approach for building a small-scale summarizer application would follow a different path compared to building an audio transcriber using transformers. In this case, the difference lies not only in the employed models, but also the necessity to use embeddings and vector stores as well. Typically, software architecture deals with designing the systems and subsystems of an application at a higher level. At this stage, a software architect need not worry about the finer details, but rather provide a sustainable architecture for a use-case. A sustainable architecture is characterized by its high scalability, modularity, and adaptability, enabling it to withstand deprecation and evolving technological demands.
A suitable architecture can significantly improve the quality of the software product. Exploring different architectures and design patterns can help solve multiple use cases efficiently. According to Butani, a software architecture pattern can help solve a recurring problem with ease, since it has already been solved before [6]. According to Bass et al, scalability is one of the attributes which is impacted significantly based on architecture and is also crucial in today's enterprise softwares. But failure in achieving scalability will prove to be unfruitful for an organisation or product. This is backed by an online article by LaFrance which stated that Twitter used to experience severe outages in early 2009, due to scalability issues. It was coined as the "Fail Whale era" [7]. Having said that software architecture plays an important role in software development, it is not the only culprit when it comes to failures, but rather could also be a very important reason for failure.
It was discussed in the introduction of how LLMs can aid and are being incorporated in the software development workflow. Since the entry into the realm of LLMs in late 2022, it would be prudent to focus on optimising LLM usage in different scenarios as it is being utilised in the industry. With the surge of OpenAI's ChatGPT, more and more organisations are adopting LLM based solutions for their projects [8]. At this point, it is ideal to explore different architectures and design patterns of

LLM integrations in systems. Based on above arguments, architectures and design patterns can significantly impact the quality of applications and thus this study would prove to be a stepping point to research in this domain. Another problem here is that there is relatively less research in the domain of architectures and patterns for LLMs.
Thus this study would help solve this problem by employing different strategies to research on architectural patterns for building LLM based systems. Identifying patterns would help to provide a structured insight into various architectures and their effectiveness in the different use-cases they are applied in. This will inturn lead to a deeper understanding of how LLMs can be effectively integrated. Additionally, this research will aid developers to create more scalable and maintainable systems. Consider a scenario where architectures are not identified for this domain, each use case would be treated as unique which would increase cost and time signficantly. Developers would have to reinvent the wheel, thus being inefficient. As mentioned above, the lack of proper architectural decisions can lead to applications being difficult to maintain and scale, which inturn would affect the reliability of the application. According to Yan, there exists seven patterns while building LLM based applications namely evals, RAG, fine-tuning, caching, Guardrails, Defensive UX and collecting user feedback [9]. On top of this, we expect to analyse the feasibility of integrating traditional design patterns in building LLM based applications and observe if traditional patterns would prove better in specific use-cases.

1.3 Purpose

The purpose of the study is to find existing architectures or design patterns that are currently being employed to build LLM based applications and analysing the effectiveness of said architectures. Effectiveness can be a measure of cost reduction, performance or even security of the applications. Examples of LLM based applications we plan to analyse include, but are not limited to, AutoGPT, a GPT based software development agent, and DroidAgent [10], a GUI testing agent. The research will make significant contributions to the field by conducting a comprehensive analysis of current architectural patterns for LLM integration. Specifically, it aims to offer detailed guidelines for best practices and highlight potential areas for future research. This will be achieved through the examination of various data sources, including academic journals, case studies from open source implementations, and developer interviews. These sources will provide a rich foundation for understanding the practical applications and theoretical underpinnings of LLM architectures. Comparative analysis will allow for the assessment of different architectural patterns in different use-cases. This will inturn help to distil common elements observed across different patterns. These methods will provide both theoretical and practical insights which can be applied in real-world settings.
The domain of this research is restricted to Software Engineering and LLMs, as we are dealing mainly with software architectures. This study could help LLM developers and companies to optimize their products, should they use LLM based solutions. The rationale for this claim would be that an architecture for a project would prove fruitful for various metrics like scalability and maintainability of the

project.

1.4 Research Questions

In response to the identified gaps in understanding architectures for LLM-based applications, this study aims to explore several key areas that have not been thoroughly investigated. The following research questions have been formulated to directly address the complexities and nuances involved in the integration of LLMs into software systems which will aid in the research's completion. The motivation for the selected research questions are provided below.

RQ1: What literature is available regarding architectures for LLM integration in an application?

RQ2: Are there identifiable design patterns used in open source software development with LLM integrations? If so, can they be described using traditional and established design patterns and architectures or are they something completely new?

RQ3: How can integrating cognitive architectures be helpful for LLM dependent applications?

RQ1 helps to establish the existing body of knowledge on LLM integration architectures. Reviewing the literature, helps to identify the key concepts, frameworks, and models that have been proposed for integrating LLMs into applications. This knowledge will be essential for understanding the state of the art and identifying gaps in research. LLMs as integrations to applications are yet to be explored in depth in the time of this research. Hence literature can include journals, articles and other grey literature sources like presentations, lectures and videos.

RQ2 would help to explore how developers integrate LLMs in their systems. This will in turn help to identify potential best practices during integration. The sole reason to be dependent on open-source systems is due to difficulty in acquiring source code access rights for commercial or proprietary software using LLMs as an integration.

RQ3 explores the applicability of cognitive architectures for LLM dependent applications. This would help reveal the efficiency of the application with said architectural concepts or if it could not be applied at all.

These research questions guide our study, aiming to provide a comprehensive analysis of current architectural patterns for LLM integration, offer guidelines for best practices, and identify areas for future research. The subsequent sections will elaborate on the methodology, culminating in the detailed objectives of this study.

1.5 Methodology

The methodology of this study starts with an extensive literature review to pinpoint the scope of the research on LLM integration. The second phase deals with examining multiple instances of LLM integration in applications to gather insights. The crucial part of the research is to identify recurring design patterns which involves a detailed analysis of the structural and functional aspects of each considered project.

These patterns are analysed comparatively to evaluate their adaptability across different use-case scenarios. To enhance the findings from the chosen projects, interviews with the developers are conducted, providing deeper insights into the practical considerations and strategic decisions involved in LLM integration. The study concludes with a synthesis of all findings, proposing a library of design patterns and observations for future LLM-based software development projects.

1.6 Outline of the thesis

This thesis is structured into several chapters. The first chapter introduces the topic, outlining the research questions and objectives, and providing an overview of the relevant literature to frame the research within the current state of knowledge. Chapter 2 deals with the theory, as it explains important concepts and various jargans which will be used throughout the document. Chapter 3 is dedicated to the methodology, presenting the research design and the approaches taken to gather and analyze data. Chapter 4 marks the core of the thesis, where we discuss the findings from the analysis, offering a detailed examination of the identified design patterns and architectural strategies employed in LLM-based applications. Each of these chapters not only delves into distinct aspects of LLM integration but also discusses the implications of these findings in light of the existing literature. Chapters 5 and 6 respectively, discuss the findings along with interesting observations made along the way of the thesis and concludes the research.

2
Background and Related Work

2.1 Concepts and definitions

This section explains some of the crucial definitions and concepts being used in this research.

2.1.1 Design patterns

Gamma, Helm, Johnson, *et al.* say that, in the domain of software engineering, a design pattern is an arrangement of components, which provides a standardized approach to solving recurring problems. A design pattern is an abstract layout, that can be applied to a set of similar use cases for optimal results. Moreover, these patterns also integrate known best practices, to aid developers in tackling common challenges in application design efficiently [11]. The authors of *Design Patterns: Elements of Reusable Object-Oriented Software* classify the patterns into three types [11]:

1. **Creational:** Consists of object creation mechanisms as seen in the Factory or Builder patterns [11].
2. **Behavioural:** Based on how objects interact with each other as seen in the Observer patterns [11].
3. **Structural:** Based on how objects or classes are structured in an application. This can be seen in the Adapter pattern [11].

For this research, Behavioral and Structural patterns are particularly pertinent, although they might not perfectly align with design patterns created specifically for LLM-based applications.

Gang Of Four's format

In the field of software architectures and design patterns, Gamma, Helm, Johnson, *et al.* coined their way of presenting design patterns as Gang of Fours [11]. The format is as follows:

Pattern Name
Classification: (Behavioral, Structural, LLM Interaction)
Intent: Purpose of the pattern.
Motivation: Explanation as to why this pattern is relevant.
Structure:

Component: Description of the component and its role in the pattern.

Applicability:

Pro-indications: Where the pattern can be applied.

Contra-indications: Where the pattern is less useful.

Benefits: Outlines the benefits of having the pattern.

Drawbacks: Outlines the drawbacks of having the pattern.

Known Uses: Projects in the analysis which made use of the pattern.

Related Patterns: Other patterns which are related to the current pattern.

This is significant as this is the format being used to present design patterns uncovered during this research.

2.1.2 Large Language Models

This section describes in detail what an LLM is, while also discussing other Generative AI-related jargon like transformers and hallucinations. This section concludes with a short discussion about artificial general intelligence and the state-of-the-art to achieve it.

What is an LLM?

According to Blank, Large language models, like the GPTs, are systems that process natural human language using deep learning. These models are trained with different objectives using large collections of text and other input material. The primary objective for these models, however, is natural language processing. The use cases for LLMs extend from linguistic prediction of upcoming or missing text in a paragraph to conversations with human beings in their preferred languages [12].

Transformers

A transformer is a model architecture for deep learning frameworks. It works by converting data into tokens of numerical values based on their relevance to each other. It works similarly, and has a similar function, to embedding models, which are discussed later in section 2.1.3.

Hallucinations

When an LLM generates incorrect, or outright nonsensical, output, that output is referred to as a hallucination [13], [14]. According to Ji, Lee, Frieske, *et al.*, there are two different types of hallucinations:

- Intrinsic hallucinations, which are generated outputs that directly contradict the source's content.
- Extrinsic hallucinations, which are outputs that are unable to be verified using the source's content. Meaning it's not always false and may be useful. But since it's not verifiable using the LLM's internal sources, it's not something one should rely on [14].

According to IBM, hallucinations can occur for several reasons, including but not limited to overfitting, high model complexity, or outdated training information [13]. Overfitting is when a statistical model fits too well to its training data, to the point where it can't make accurate predictions about non-training data [15]. Hallucinations are a very big validity threat to LLM outputs. As such, applications making use of LLMs would need to minimize them. One solution to this may lie in architectures and patterns.

Artificial General Intelligence

In a general sense, an artificial general intelligence (AGI) is an artificial intelligence that is as, or more, proficient than humans at a range of tasks [16]. More specific definitions are heavily disputed among researchers, leading to the term being rather vague.

LLM Agent

Agent is a common term used in the context of artificial intelligence and robotic process automation. But in the context of LLMs, an agent is an autonomous software entity that makes use of LLMs to perform challenging tasks [17]. The way the agents solve problems differ between use cases and implementations. Some agents may require the use of external tools, or some kind of planning routine for larger problems. Agents can also be used together to simulate different people having different occupational roles in a simulated team, of sorts. Other cooperative structures also exist, they are not required to simulate humans or human group structures.

2.1.3 Generative AI Applications

This section is about Generative AI (GenAI) and common components of applications using them, at least according to Choi. They are embedding models, vector databases, data filters, and prompters. However, Choi's claims about these components may be limited to LLM based applications rather than all types of GenAI applications. The concepts in-context learning and retrieval augmented generation are also included here. Both of these are used to minimise inaccuracies and hallucinations in GenAI applications.

What is classified as GenAI?

GenAI is simply put any AI model that is made to, or able to, generate original content, regardless of the AI technique used to generate the content [19], [20], this includes LLMs for example. This is opposed to, for example, discriminative models, which according to Foster, are models that are trained to estimate the probability of observations belonging to specific categories or labels[20], [21].

Embedding Models

Embedding, in the context of machine learning and by extension AI, is representing non-numerical data (like text, images etc.) as numerical coordinates in a vector space

[22], [23]. The value of the numerical points are based on how semantically close the original data points are. This is to make the data usable by machine learning algorithms. An example would be the three words "bad", "dad", and "father". Purely from a data perspective, "dad" and "bad" are closer since they share 2/3 of their letters. Whilst from a semantic perspective "father" and "dad" are closer, since they're synonyms. Another example:
There are three pictures: One of a brown dog with a blue background, one of a grey dog with a green background, one of a brown cat with a blue background. Whilst the first and third pictures are more similar based purely on the pixels, the first two pictures are closer semantically, both of them being dogs. At least if the desired outcome is to classify taxonomic rather than chromatic similarity. This is the purpose of an embedding model, it's a data structure that embeds data, as previously described, to make data points related based on semantic significance rather than letter or byte similarity. Embeddings, and embedding models, are essential to GenAI as it's the main way they receive and process information.

Vector Database

A vector database is, as the name suggests, a database that stores vector data. It is this type of database that the vectorised data from embedding models are stored and retrieved by the AI [18]. In the context of a GenAI application, the vector database doesn't necessarily have to be open to new data, it just has to be combine its data with vectorised input data that has gone through an embedding model to give the input data context. Meaning that the vector database in question only contains training data.

Data Filter

A data filter in this context is one that filters sensitive data to make sure that the model doesn't process or output said data [18]. Sensitive data being things like personal identifiable information, passwords, etc.

Prompter

A prompter, or prompt optimisation tool, is a tool that optimizes the final prompt before it's sent to the LLM. One common type of prompter does this by packaging the initial prompt, or query, with the necessary, or at most least relevant, context [18]. The context comes in the shape of embeddings and are retrieved from a vector database. This means that this type of prompter requires some kind of embedding model and vector database.

In-Context Learning

In-context learning is a prompt engineering technique in which a language model is taught new tasks via examples with answers, allowing the language model to infer solutions to new problems based on the examples [24], [25]. This allows language models to be used for purposes they were not initially trained for. It also avoids compromising the stability of the model's parameters since they rarely persistently

store user inputs. This approach has become very common for building LLM based applications as it doesn't require fine-tuning or training of LLMs.

Retrieval Augmented Generation (RAG)

Retrieval augmented generation is when an LLM doesn't just rely on its training data to relay information, as that information can be outdated, or false if the training data included false information [26]. Instead it retrieves information from some kind of separate data storage, like the internet or something more limited. This means that the LLM now relies on more than just its training data, making it less likely to give wrong information, like hallucinations.

2.1.4 Software Architectures

Software architectures are, in a way, blueprints for software systems, describing high level structures and how they interact [27]. Being abstract representations of the overall structure of software systems, they omit practical details such as algorithms and implementation details, focusing instead on design patterns and less specific, externally observed, properties. According to the IEEE Computer Society, software architectures, and the study of them, are integral to the software engineering field.

Why focus on software architectures?

Since GenAI and LLM based applications are part of the software engineering field, the concept of software architectures and design patterns should apply to them as well. However, established architectures an patterns may not be very applicable to these new types of applications. And since they have exploded in number the last few years, one would expect developers to come up with their own architectures to solve problems that arise when building their applications, as opposed to just having a messy codebase.

Blackboard architecture

According to Hayes-Roth, the blackboard architecture is a problem-solving framework which is used in AI systems. In this architecture, different knowledge sources are integrated using a collaborative approach. A shared data structure called the blackboard is made use of for the integration part. In summary, multiple subsystems or smaller knowledge sources contribute to developing a solution [28]. The significance of this architecture will be explained in the Discussion section (Section 5.4) in the document.

Scalability

Scalability, in terms of software, has two different uses. One is load scalability, which is the ability of a software system to handle increased amounts of work without suffering significant performance losses [29]. Examples of increased amount of work include: A website needing to handle more traffic. A database needing to store ever increasing amounts of data. The other type of scalability is architectural scalability,

which is the ability of a system to be expanded in a chosen dimension without needing to perform major changes to the system's architecture. Performing these changes every time expansion is necessary can become very costly in the long run. Both of these uses are relevant to this research as both of them are important costs to consider when developing larger software systems, LLM ones included.

2.2 Related work

This section discusses the state-of-the-art in terms of GenAI architectures and design pattern considerations from various sources.
A blog by Choi, provides more potential patterns to effectively integrate LLMs into an application. They suggest techniques such as RLHF and Fine-tuning on top of the already mentioned In-context learning, which could help direct the application towards a more specific use case [18]. Another article by Li, Zhang, Yu, *et al.* further justifies the validity of the multi-agent architecture with their claim being that more agents is what is required for efficient functioning of an LLM based application [30]. Github released a blog discussing potential architectures for LLM based applications. Here they discuss various techniques employed to improve performance of such applications versus the quality of output. Furthermore, they discuss potential architectures which can optimize the functioning of an LLM based application [18]. Ryu, mention a potential design pattern for LLM based applications, which they got by interviewing the heads of several AI based startups [31]. The authors Heiland, Hauser, and Bogner, provide an overview of design patterns for AI-based systems. According to their study, there were a total of 34 new patterns and 36 traditional patterns which could prove effective in applications. Since one of the phases in our study includes the usage of cognitive architectures for LLM based systems, we decided to do some general research in that context. Their pattern called in-context learning, is broken down into a few phases namely embedding, retrieval and inference, which help to optimize LLM usage in their applications [32].
An article by Tung, demonstrates different architectural considerations to take into account while building GenAI applications [33]. Moreover, another article by Yan, discusses different purposes of different design patterns which can be employed while building LLM based applications or systems.

3
Methods

This section discusses the methodology adopted in this research to explore design patterns in LLM dependent applications. By employing a mixed-methods approach, this study aims to not only identify and analyze existing patterns but also to contribute to the theoretical and practical understanding of integrating cognitive architectures with LLM applications. The adoption of a mixed-methods approach in this study is strategically designed to address the complex and nuanced nature of LLM integration within software applications.
The investigation commences with a systematic literature review to establish a foundational knowledge base and to identify gaps in current research. This initial phase is integral for framing the subsequent analysis and for ensuring a comprehensive understanding of the field. Section 3.1 describes the exact methodology employed in this research phase and this would help answer RQ1.
Section 3.2 involves the identification of design patterns in LLM-dependent applications through a two-pronged strategy: firstly, project identification, which rigorously selects LLM-utilizing projects for analysis followed by a qualitative analysis, enriched by developer interviews and a review of project documentation, provides depth to the quantitative findings, allowing for a nuanced understanding of the motivations behind architectural choices and their implications on LLM applications, thus answering RQ2.
Finally, in section 3.3, the methodology culminates in an evaluative phase, as traces of cognitive architectures are analysed in the identified applications. This evaluation is descriptive but seeks to ascertain the practical and theoretical contributions of cognitive architectures to LLM dependent applications, thus answering RQ3.
Visually, the methodology has been presented in a flowchart format in figure 3.1.

3.1 Identifying relevant literature

This research involves a scoping review to get an idea of the range of research in the domain of Generative AI, LLMs and cognitive architectures. The initial selection for papers were done by going over titles of the studies and articles. To conduct a thorough and targeted literature scoping review, clear inclusion and exclusion criteria were defined. These criteria aim to select the most pertinent and credible sources on the integration of Large Language Models (LLMs) within software architectures, with a focus on recent developments and key contributions to the field. The inclusion criteria prioritise studies that offer insights into design patterns, architectural frameworks, and case studies involving LLMs. Conversely, the exclusion criteria are

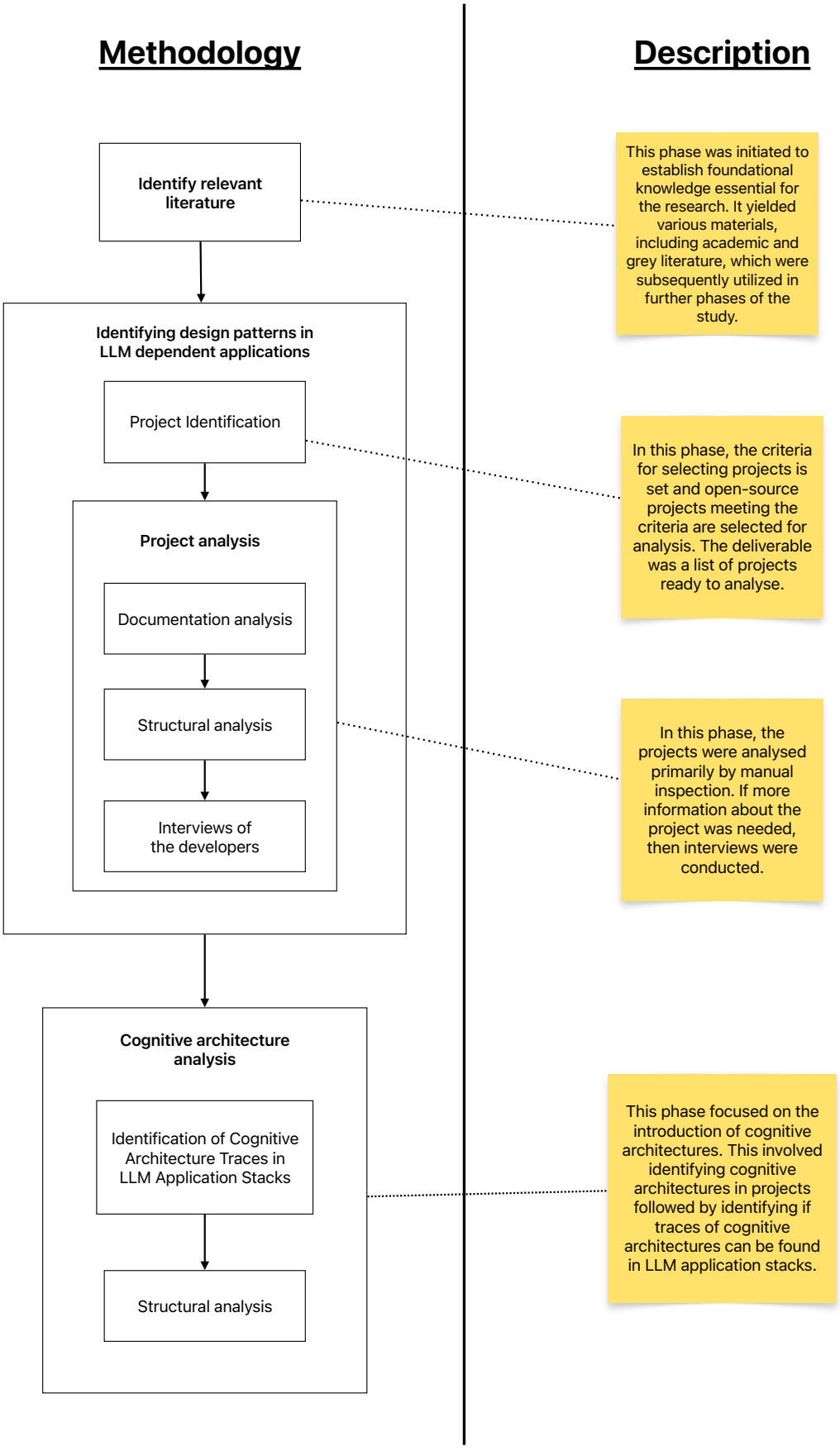

Figure 3.1: Research methodology

designed to deprioritise, or even eliminate, irrelevant or outdated works. The criteria are not hard classifiers, they are loose guidelines with some criteria can override other criteria. A criterion can have one of three levels of importance:

1. **Major:** Very important to fulfill to be included/excluded.
2. **Moderate:** Moderately important to fulfill to be included/excluded.
3. **Minor:** Not especially important to fulfill to be included/excluded.

This means that sources that fulfill exclusion criteria aren't automatically disqualified, they may just require some more scrutiny depending on the criterion's level of importance. The specific criteria are detailed in Tables 3.1 and 3.2 below.

Table 3.1: Inclusion Criteria for Literature Review

ID	Criteria	Description	Rationale
I1	Relevance and Scope (Major)	Focus on works that discuss LLMs in the context of software architecture, and/or cognitive architectures including design patterns and case studies.	Ensures the review focuses on the primary area of interest.
I2	Recency (Moderate)	Give preference to publications from the past five years.	Ensures the information is current and relevant.
I3	Source Reliability (Major)	Select peer-reviewed articles from well-known academic journals and conferences.	Guarantees the sources' reliability and validity.
I4	Significance and Impact (Minor)	Include sources with high citation counts or from influential platforms.	Highlights significant contributions to the field.
I5	Open Source Repositories (Minor)	Include publications that propose LLM-based applications or tools with open repositories.	Useful for further analysis and practical application.

Table 3.2: Exclusion Criteria for Literature Review

ID	Criteria	Description	Rationale
E1	Irrelevance (Major)	Exclude works not focused on LLMs or software architecture, and those addressing non-architectural aspects.	Keeps the review focused on relevant topics.
E2	Outdated Publications (Moderate)	Exclude articles older than five years unless they are highly cited.	Ensures the review includes contemporary research.

Table 3.2: Exclusion Criteria for Literature Review

ID	Criteria	Description	Rationale
E3	Low Impact (Minor)	Avoid sources with low citation counts or from less influential platforms.	Focuses on impactful and recognized research.
E4	Language Consistency (Major)	Exclude publications not in English.	Maintains consistency in the review process.

Table 3.3: Exceptions Criteria for Literature Review

ID	Criteria	Description	Rationale
X1	Historical Significance	Include older publications that are famous in the field irrespective of how old they are.	This ensures foundational research with significant impact on the field is included.
X2	High Quality with Limited Citations	Include studies with low citation counts if they exhibit high methodological rigor and relevance.	This ensures that high-quality research is not excluded solely due to low citation numbers.
X3	Studies from other fields	Include studies are not from the same domain as the current study. For example, studies which connect human cognition to cognitive architectures can prove useful to understand the capabilities of cognitive architectures better.	This expands the review to include beneficial perspectives from related disciplines.
X4	Grey Literature	Includes studies addressing intriguing topics that are not published as traditional academic publications but rather as online articles, seminars, or presentations.	Given the blooming stage of the Generative AI domain, a considerable amount of significant literature is published through non-traditional mediums, thus increasing the chances for us to gather more insight.

Furthermore, in addition to screening papers for inclusion, the scientific quality of the selected studies were assessed. This was done based on quality attributes in the study such as quality of rigour in research design and methods employed and bias towards an organization. This formal assessment conducted independently by both

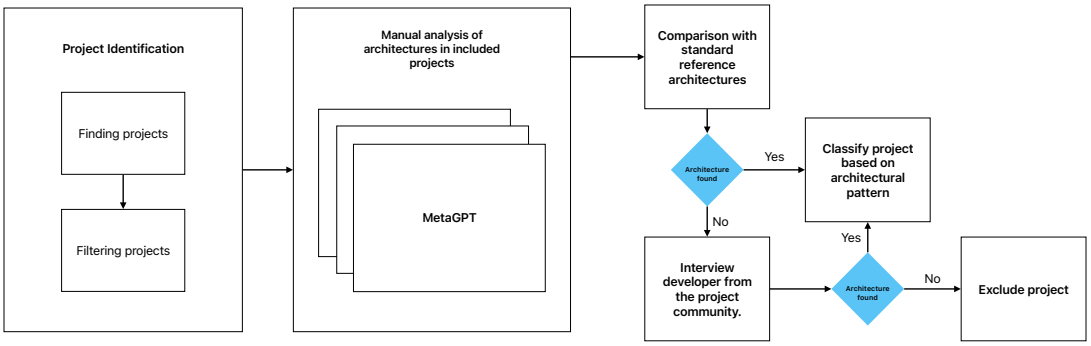

Figure 3.2: Project identification

of us, helps to refine the studies to include in the final sample, determine whether or not the differences in quality may affect their conclusions. No quality scores or other quantitative metrics were employed to accept or reject a study.

3.2 Identifying design patterns in LLM dependent applications

The following section describes the methodology employed in attempting to answer RQ2, which helps to identify existing architectural patterns for LLM dependent applications. The subsections 3.2.1 and 3.2.5 focus on methods employed in identification and analysis of open-source projects in detail.

3.2.1 Project identification

Identifying and analysing open-source projects form a major part of the data collection process. The project identification phase of the research relied primarily on internet blogs, academic publications about an application and GitHub searches. The source code of the projects are limited to GitHub. The following sources of online articles and publications, help in identifying different LLM dependent projects. The projects mentioned by Van Vaerenbergh were considered, as it was a well curated list of open source LLM dependent projects with links to GitHub repos for the projects [35].
The project identification process involved a top-down approach involving two subphases. Figure 3.2 displays the entire flow of selecting a project.

1. Finding candidate projects
2. Filtering candidate projects

3.2.2 Criteria for finding projects

The following criteria are being used for finding projects included in this research.

Relevance to LLMs

The project should utilize or be relevant to LLMs, either using APIs or in other similar forms.
Motivation This ensures that the study is focused on the primary area of interest and that the deliverables are relevant to the domain of LLM dependent applications.

Openness to Contributions

The project should be open to contributions. Although open-source insinuates the need for contributions from external developers, the level of adaptability of the application itself plays a major role in identifying the architecture of the application.
Motivation This emphasizes the practical aspect of architectural adaptability and the collaborative nature of development in the open-source community.

Community Engagement

The project should have at least 1000+ stars or forks on GitHub. This is a measure of community engagement, as a large and active community might come of use, if an interview with the developers is required.
Motivation A vibrant community not only signifies the project's impact and acceptance but also increases the likelihood of gaining valuable insights through developer interviews or community discussions.

Output Classification

The project can be classified into one of the following categories based on the outputs achieved from the project:

1. **Text-generative** - The project generates text as the desired output.
2. **Translative** - The project translates text and provides the translation as the desired output.
3. **Code-generative** - The project generates code based on input prompts as the desired output.
4. **Image-generative** - The project generates one or more images based on an input prompt as the desired output.
5. **Hybrid** - The project can be classified as hybrid if it performs one or more of the above mentioned functions.

Motivation The classification of projects based on the type of output allows for a structured analysis across different application domains of LLMs. A detailed motivation on the selection of the categories is given below.
The categories delineated in the above subsection, in item 4, broaden the scope for identifying relevant applications for analysis. Text-generative LLMs are an integral category as according to Marcus, Leivda, and Murphy, LLMs are primarily designed to understand and generate human language [36]. This inherent functionality of LLMs not only aligns with their core technological objectives but also highlights the vast potential for practical applications, making it an essential focus for our study. Translative provides another narrow domain for LLMs to understand and convert text to different languages, thus providing yet another use-case to focus

on. Code-generative is an area of interest due to its potential to streamline software development processes. Image-generative helps to expand the usage of LLMs beyond text-based tasks. Finally, the hybrid is a category for the applications which do not fit into a single category, but into multiple categories.
Once the projects have been found following the criteria mentioned in the previous subsection, the selected projects are filtered based on an exclusion criteria as mentioned in the following subsection. As with the literature, the project exclusion criteria are not meant to automatically exclude a project, with some criteria being more severe than others.

3.2.3 Criteria for Filtering Projects

The following criteria are being used for filtering the projects that were found using the criteria in the previous subsection.

Lack of Architecture

The project meets the inclusion criteria but is designed without using an architectural pattern.
Motivation This ensures that the study's results are consistent. Furthermore, the lack of architecture in a project provides no aid to this study.

Lack of Community Engagement

The project meets the inclusion criteria but should an interview with the developers be needed, the following list establishes additional exclusion criteria:

1. If the developers ignored the posted questions due to uncertain reasons.
2. If the developers did not wish to answer the question fully for reasons including lack of knowledge regarding the concept being specified, lack of sufficient explanatory material from their side, and business secrecy.
3. If the project doesn't have any kind of public forum or similar.

Motivation Insufficient community engagement can present a significant challenge, as it impedes the acquisition of desired outcomes.

Smaller Size and Complexity

The project meets the inclusion criteria but is not large enough to make use of an architecture or design pattern.
Motivation The requirement for a project to be "large enough" serves to filter out trivial or overly simplistic projects that lack a sophisticated architecture or design pattern.

Project Classification as a Library

The project should demonstrate practical applications of LLMs rather than function primarily as an LLM-based library.
Motivation This criterion is aimed at understanding how the LLMs are integrated into solutions that address tangible problems, thus providing a perspective on the

pros and cons of current LLM dependent applications. Moreover, architectures for libraries and applications tend to be very different, making library projects out of scope.
Having posed an inclusion-exclusion criteria for identifying projects, there can also be some exceptional criteria for choosing outlier projects, which can pose a discussion about the state-of-the-art albeit not fulfilling the inclusion criteria.

3.2.4 Data Collection

The primary data for this research is derived from manual inspection and interviews. Although static code analyzers and dependency graphs were initially considered, they were ultimately deemed unnecessary, as the required data was effectively obtained through manual inspection and interviews. Manual inspection enabled the identification of architectural flows, which proved crucial to the relevance and accuracy of the research. Initially, the tool Codesee was utilized for this purpose until it became defunct. Thereafter, Visual Studio Code, equipped with the file tree extension, was employed to manually inspect the source code. This approach facilitated a comprehensive analysis of the structural and architectural aspects of the repositories, ensuring the collection of relevant and detailed data.

3.2.5 Analysis

A majority of the research relied on analysing open-source projects to identify their architectures. The goal of answering RQ2, is achieved by exploring the codebases of several LLM based projects[1].
The project analysis encompassed comprehensive data collection to discern the architectural framework employed by the open-source application. This process entailed constructing dependency diagrams, collaboration with developers through interviews to outline high-level architectural diagrams for better understanding of the system operations, and examination of project documentation to evaluate efficiency metrics.

Structural Analysis

The structural analysis aims to scrutinize the architecture of various open-source repositories used in LLM-based applications. This involves a detailed examination of the project's files to identify API calls to LLM providers, the utilization of agents, and the interaction patterns among these agents within the codebase. The analysis focuses on class hierarchies, method usage, and other LLM-specific entities through manual inspection. Initially, a parsing visualization approach was considered for this analysis. However, this approach was subsequently abandoned due to the complexity and lack of coherence observed in various source codes. The initial results obtained from MetaGPT necessitated a pivot in our methodology to exclude visualizations. Figure \ref{fig:codesee-metagpt} illustrates a segment of the dependency graph generated by MetaGPT.

[1]In the current context, LLM based projects imply LLMs being a central part of the project, hence providing the features displayed by the projects.

Selection of LLM Providers

The study encompasses a range of LLM providers, including:

- OpenAI [37]
- HuggingFace [38]
- Anthropic [39]
- LangChain [40]

Scope of Analysis

Given the general practice of making API calls in the backend of applications, the analysis focuses solely on the backend code, rendering the UI components (e.g., React, Angular) irrelevant to this research.

Identification of LLM-Specific Entities

The following LLM-specific entities are identified and analyzed within the applications:

1. **Use of Agents**: The presence and role of agents in the codebase.
2. **Agent Interaction**: The patterns and methods of interaction among agents.
3. **VectorDB**: Integration and utilization of vector databases.
4. **Embedding Model**: The embedding models used within the applications.
5. **LLM Instance**: Instances of LLMs and their configurations.
6. **LangChain Agents**: The implementation and behavior of LangChain agents.

Each identified entity is meticulously examined to understand its implementation, functionality, and contribution to the overall architecture of the application.

Reference architectures

The process of extracting architectural patterns from application codebases can be made efficient through the normalization of architectural components to a select number of reference architectures. To achieve this, pre-established LLM application architectures were evaluated for their utility in identifying shared components between reference architectures and the projects under investigation.

The reference architectures were primarily selected based on technology blogs from reputable industry leaders, that recommend the usage of the architecture. The architectures that were mentioned in different academic papers for different use cases were also considered for reference. Using this approach, four reference architectures were found. Projects were then categorized according to these four reference architectures. This categorization process should result in projects being directly mapped to a singular reference architecture or incorporating elements of one or more reference architectures within their design. The following reference architectures were chosen for analysis in the subsequent phase:

Table 3.4: Parent LLM architectures

S.No	Architecture	Motivation	Figure
1	Retrieval Augmented Generation (RAG)	The reason for selecting this pattern is because it was accepted by HuggingFace, GitHub and other leaders in the domain, thus indicating its popularity and acceptance among developers.	Figure 3.3
2	In-context learning	This is relevant because the reference architecture proposed, consists of the most commonly used tools in design patterns by major AI startups and tech companies [31].	Figure 3.4
3	Ad-hoc	This is the simplest architecture for LLM based applications and is an architecture based out of API calls to instantiate LLMS and uses a technique called Prompt engineering. Here, the user crafts a prompt to control the output derived from an LLM.	Figure 3.5
4	Multi-agent	Multi-agent architectures are among the widely spoken architectures in the LLM space. They are more suited for solving complex and distributed problems where individual components or agents can either act together or separately to achieve a common goal.	Figure 3.6

According to an article by HuggingFace, implementations of RAG have shown significant improvements in generating more factual and diverse language outputs [41]. This aligns with the need for architectures that can handle the dynamic content requirements of LLM applications. The architecture is depicted in the figure 3.3 where the various components which form the RAG architecture are shown. Moving on, an article published by Oracle discusses how in-context learning improves the relevance and accuracy of responses by LLMs, particularly in customer interaction scenarios, making it invaluable for maintaining high user satisfaction and engagement [42]. According to Ryu, figure 3.4 depicts another similar RAG LLM application architecture that was chosen after consideration [31]. Figure 3.5 displays an architecture where the application interacts with LLMs using the OpenAI API. An article by Azure discusses how APIs simplify the access to LLM technologies, thereby broadening their application across various platforms and making the integration process more manageable and accessible [43].Furthermore, a multi-agent architecture for LLM dependent applications was also proposed which is also considered in this research for an application architecture. This architecture was discussed in an article

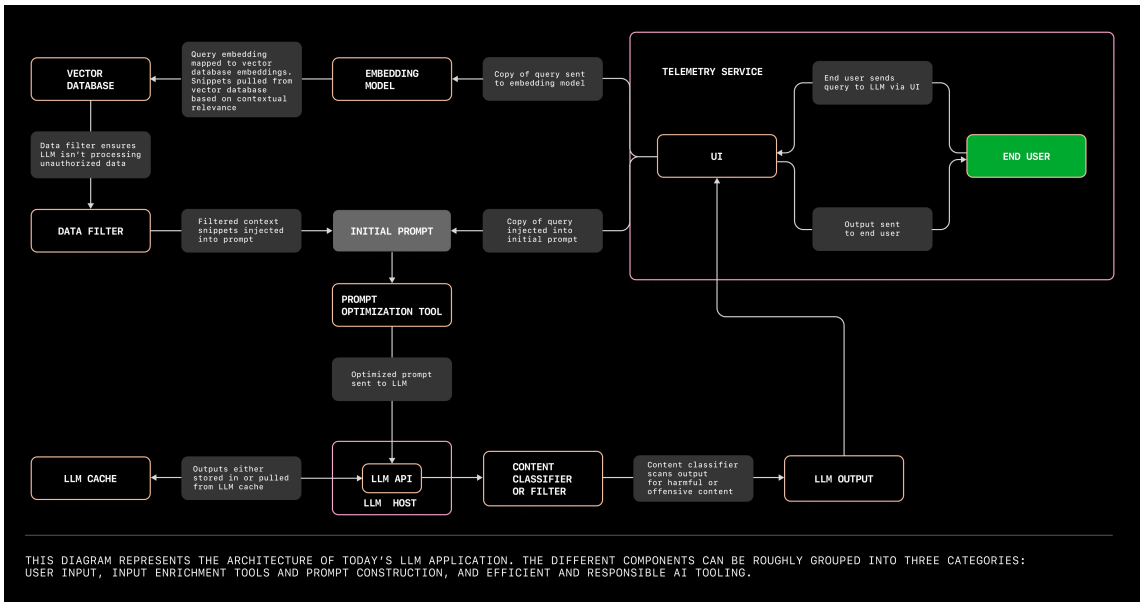

Figure 3.3: RAG LLM application architecture (GitHub) [18]

by Azure GitHub saying that the collaborative and decentralized decision-making capabilities of multi-agent systems make them suitable for complex environments often addressed by LLM applications, as they allow for flexibility and dynamic response generation [44]. According to Greyling, RAG is an architecture which focuses on improving accuracy of the results, where in-context learning helps by serving a niche scenario for better context awareness [45]. Finally, multi-agent architecture helps to create multiple agents which can act in co-ordination and help with the scalability of the application.

Multi-agent architecture in itself is a high level definition. Rather this architecture gets its granularity based on how the agents interact with each other. There are certain ways agents can communicate or coordinate with each other to achieve a common goal, which will aid this research to identify different patterns in projects. An assumption included here would be that the derived agent interactions could give rise to certain design patterns in different applications. Observations of relevance to the research include agents interacting with each other in different formats (cyclic, autonomous etc.), usage of different smaller architectures for each agent.

The major concern of the application architecture would be the interactions between the LLMs and the application logic. Using this information, a simple architecture was also constructed which could be used in building prototypes of the applications. This is depicted in the figure 3.5. Moreover, the figure also provides a high level understanding of the application in itself. The front-end component in the figure handles user interaction, while the backend API handles the business logic. During the time of writing this paper, LLMs are being offered as services for a usage fee by providers like OpenAI [37], HuggingFace [38], Langchain [40] and Anthropic [39]. An article by Wrba, explores how an individual can leverage OpenAI LLMs using APIs into different applications [46].

After careful analysis, it was decided that the aforementioned application architectures can be considered as common practices and would also highlights the flexibility

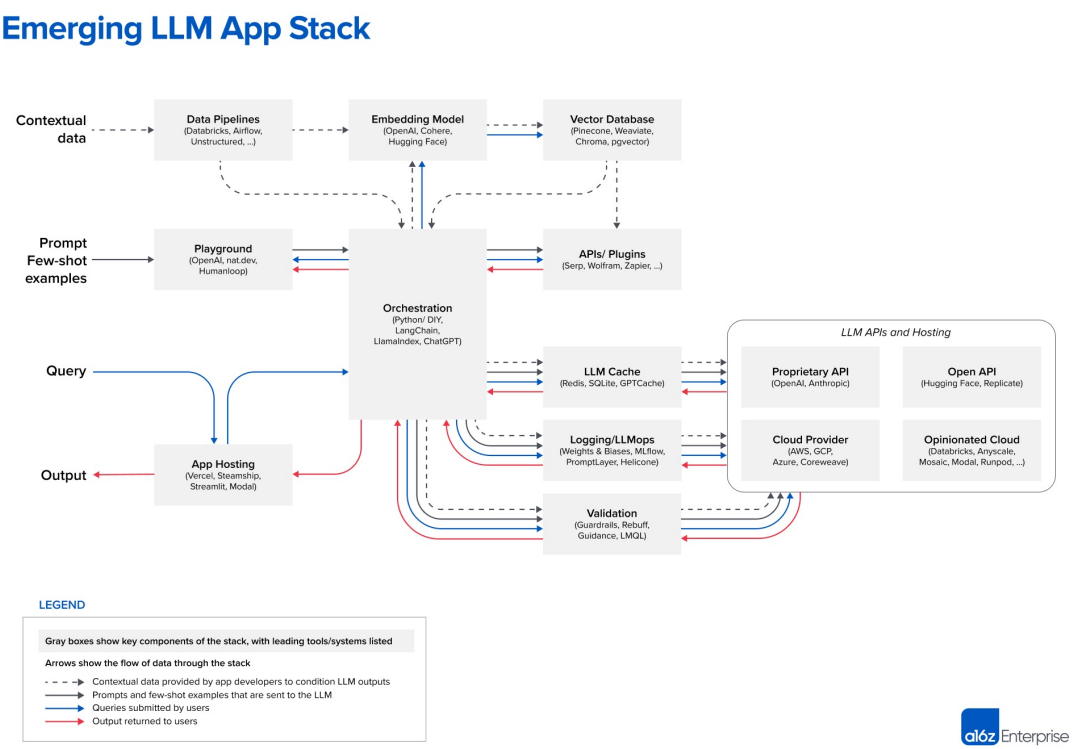

Figure 3.4: In-context learning LLM architecture [31]

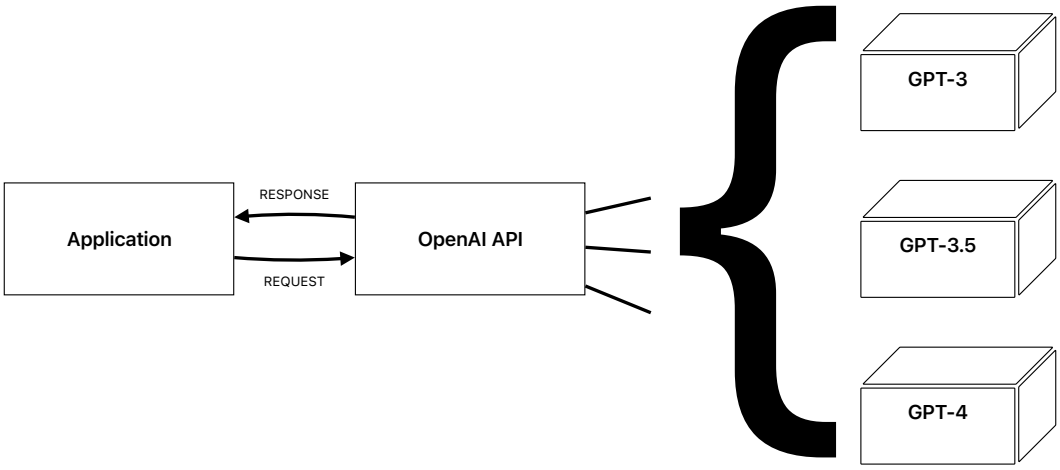

Figure 3.5: Ad-hoc LLM architecture using OpenAI API as an example.

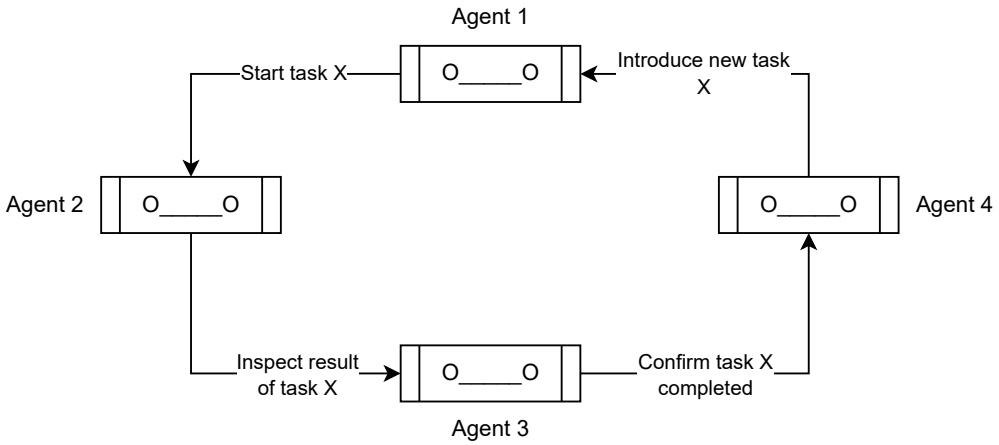

Figure 3.6: Multi-agent LLM architecture with cyclic workflow

of an application built using either architecture. Although several components are presented in the application architectures, an application can function without several of the components mentioned in the architecture. Moreover, the usage of a standard architecture diagram simplifies the complexity of comparing multiple systems by focusing on a singular set of key components and their interactions in the system. It would also help to identify patterns and best practices.

Interview

In case of absence of results from manual searching for the architectures, the community of the open-source project was contacted and an informal interview with the developers was conducted. The sole purpose of the interviews is to improve the comprehensibility of the project, in case of ambiguity. In the current context, ambiguity can mean lack of proper documentation, failure to identify a design pattern, and lack of proper explanations for components. Thus the questions to developers were phrased in the following manner with a few personalized changes,
“ Hello, I am a Master’s student in Computer Science, currently researching some things relating to applications and projects using LLMs, and I’m wondering if there may be a graph, or similar, showing how [**application**] works, in an abstract fashion. Or maybe some document describing the flow or abstract structure? This is research about finding possible design patterns and architectures for said LLM based applications and projects. If there aren’t any visuals, like graphs, available, would it be possible for a description of it from which I could draw something up? “
The interviews were conducted through text exchange through platforms like Discord and Twitter. After the aforementioned text was sent, the conversation takes a semi-formal approach as each project required attention in different subareas. The responses of the people approached, were recorded and the people were anonymised for the sake of privacy.

3.2.6 Outlier projects

This subsection explores the distinct analytical methodology employed for projects that deviated from the conventional approach. These exceptional projects, chosen according to specific inclusion criteria, are characterized by their remarkable content, which provides valuable insights and enhances the overall discourse.

MetaGPT

Initially, an online service CodeSee (*defunct since 22-02-2024*) was used to visualise the code base using the dependency map feature.
Furthermore, the developers of the application were consulted over Discord to gather more intel about the flow of the application. Moreover, *MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework* a publication by Hong, Zhuge, Chen, *et al.* was analysed to observe the working of the application [4].

Promptify

Promptify is a prompt generation tool [47]. It is designed to convert natural language inputs into prompts, which structure the output in a specific way suited to particular niches. The structure and the prompt are based on a pre-defined template. An example of the functioning of Promptify is displayed in the figure 4.4. Using this, the code and the documentation were observed. An attempt to contact the developers was also considered, but was deemed unnecessary due to their Discord server being inactive.

3.3 Evaluating cognitive architectures in LLM dependent applications

Cognitive architectures serve as a foundational framework for simulating human cognition within software systems [48]. This concept is crucial for understanding how these architectures can enhance LLM applications by providing more robust, adaptable, and intelligent functionalities. The inclusion of cognitive architectures goes beyond theoretical exploration, directly contributing to practical outcomes in design patterns and application efficiency. The research methodology for this study is designed to comprehensively examine the utilization and implementation of cognitive architectures within various applications. The methodology is structured into several key phases, each contributing to a holistic understanding of how cognitive architectures influence and drive the functionality of complex systems. Answering this research question would require three steps namely, identifying existing cognitive architectures, identifying aspects of the cognitive architectures in existing architectures discovered from RQ2 and finally an analysis of the results. The cognitive architectures are explored in the context provided by the reference architectures. This means that the general guidelines gained from reference architectures are applied and tested against the more complex requirements of cognitive architectures. This step-by-step approach ensures that the study not only explores each

architecture independently but also examines how they can be synergistically used to enhance the design and functionality of LLM-based applications.

3.3.1 Analysis

Following the literature review, the methodology shifts focus towards the practical application of these theories. Applications derived from RQ2 are analysed in-depth to identify traces of cognitive architectures. The analysis was conducted in two parts namely structural analysis and document analysis.

Structural analysis

This analysis differs from the one detailed in Subsection 3.2.5, as it specifically examines cognitive architecture entities within the applications. The entities to be observed are identified based on a comprehensive literature review. The list of entities to be observed derived from the literature review, ensuring that the selection is grounded in existing research and aligns with the objectives of this research. The selection of these entities is guided by the following criteria:

1. **Common Entities:** Entities must be prevalent across multiple cognitive architectures. This ensures that the selected entities are representative and relevant across different frameworks.
2. **Similarity to LLM-based Application Architectures:** The chosen entities must share structural or functional similarities with components of the observed design patterns in LLM-based applications. This criterion ensures that the entities are pertinent to the context of LLM architecture.

Documentation analysis

In addition to the structural analysis, we conducted a thorough examination of the documentation associated with each project to identify references to cognitive architecture usage. This analysis aims to complement the structural findings and provide a more comprehensive understanding of the design patterns employed in LLM-based applications. The following steps and criteria were used for the analysis:

1. **Document Collection:** All available documentation for each project was collected, including README files, wikis, official documentation, and developer notes.
2. **Inclusion criteria:** Documentation must be officially associated with the project and accessible through recognized channels. Explicit mention of cognitive architecture usage was welcome but not required.
3. **Analysis:** Findings from the documentation were cross-referenced with results from the structural analysis to ensure consistency and validate observations.

3.4 Deliverables

The following section describes the deliverables from this research which can be useful in answering the research questions.

3.4.1 Architectural flow diagrams

The visualizations of architectures of the different projects that were analysed contribute as a deliverable for this study. These visualizations help readers to understand the flow of the application and the significant components being used in each project.

3.4.2 Classification of projects

Answering RQ2 involved the classification of architectures of different projects with additional source code metrics which were discussed in the results. In the current context, the metrics were primarily architecture related metrics like Scalability index etc. The projects were classified based on the following,

1. Nature of the output.
2. Identified architectural pattern.

3.4.3 Library of design patterns

The major contribution of this research is to provide a comprehensive library of design patterns framed in the Gang of Four's format, specifically tailored to LLM-based applications. This library will serve as a valuable resource for developers and researchers, offering standardized and well-documented solutions for common design challenges in the context of LLM integration. By leveraging established design principles, this research aims to facilitate the development of robust and scalable LLM-based systems, hence enhancing the overall quality of software architecture in this emerging field.

3.4.4 Cognitive architectures

Observations of the usage of cognitive architectures across different projects are meticulously recorded. This involves not only identifying the presence of cognitive architecture components but also understanding their specific implementations and roles within each project.

3.4.5 Interviews

The interviews that were conducted were recorded and anonymised to safeguard the privacy of the interviewees. Moreover, the queries posed to the interviewee and their responses were tabularized and presented in detail in the Appendix.

3.5 Validity and Ethical Considerations

The following section highlights the various ethical considerations taken during the course of this research and the validity of the research. The ethical challenges must be addressed to ensure responsible and equitable use of the data collection in this research.

3.5.1 Validity

The validity of this research is upheld through a rigorous methodology that includes interviews with actual developers, ensuring that the analysis accurately reflects practical implementations and expert insights. The use of firsthand accounts from developers provides a strong foundation for the reliability and relevance of the findings.

The methodology employed in this research is designed to be replicable across different projects, thereby maintaining its validity. However, it is acknowledged that the results may vary due to the active and evolving nature of the repositories analyzed. As these projects are continuously developed and updated, the specific outcomes of similar analyses may differ over time. This variability highlights the importance of conducting longitudinal studies and re-evaluations to capture the latest developments and trends in LLM-based applications.

Furthermore, to account for potential discrepancies caused by updates to the repositories, the research includes mechanisms for version tracking and documentation of the specific versions analyzed. This ensures transparency and allows for a clear understanding of how updates might influence the results.

In summary, while the core methodology remains valid and robust across different projects, the results are inherently subject to change due to ongoing development. This dynamic aspect is an integral part of the research, underscoring the need for continuous monitoring and adaptation.

3.5.2 Licensing and Credits

The projects examined in this research were open-source, which introduces specific ethical considerations. Respecting the original creators' licenses and providing proper attribution is crucial when using open-source projects. Furthermore, any alterations or enhancements made to the open-source code should be shared with the community under the same open terms. Ethical use of open-source projects also requires transparency in reporting findings and making the research accessible to the broader community, ensuring that the research benefits are widely distributed.

3.5.3 Privacy and Consent

The use of interviews in research raises ethical concerns regarding the reuse of collected data. To address these concerns, all conducted interviews were anonymized, and explicit consent for data reuse was obtained from participants after thoroughly explaining the scope and purpose of the research. Participants were informed of their right to withdraw from the study at any time. The anonymization process involved removing any personally identifiable information and using coded identifiers to maintain confidentiality. This approach ensured that participants' privacy was protected throughout the research process.

3.5.4 Ethical Use of AI

In this research, helper tools such as ChatGPT were primarily employed for tasks like spell checking, plagiarism checking and language refinement. In places, ChatGPT was also used to paraphrase text to make it sound more formal. Importantly, LLMs were not used to generate or influence the actual content of the research itself. This ensured that the core findings and analyses remained entirely the product of original thought and investigation, maintaining the integrity and authenticity of the research.

4

Results

This chapter discusses the results observed after performing the methodology. Since this is a qualitative research study, it primarily involves analyzing patterns and themes rather than focusing on quantitative metrics or statistical data. The results being analysed are in the form of diagrams and discussions which help to interpret the results better.

4.1 Introductory overview

During the course of the research, fifteen projects were identified for potential analysis. However, only nine were accepted and analysed, with six being rejected for differing reasons. These nine were deemed enough to answer the first part of RQ2, specifically: "Are there identifiable design patterns used in open source software development with LLM integrations?" With these results, which are showed in section 4.4, in tandem with the results from section 4.2, the second part of RQ2 is also answered. RQ1, on the other hand, was answered thoroughly in section 4.2. RQ3 is also answered by combining parts of the results from section 4.2 and section 4.4, which answered RQ1 and RQ2 respectively. In further sections, the results are organized as follows. Initially, the findings from literature review are discussed, which along with the patterns observed upon structural analysis of the projects give rise to a library of LLM based application design patterns which are described in the GangOfFours model. This is the primary contribution of this research. Then the results from the structural analysis of the projects are discussed in detail followed by the research on cognitive architectures.

4.2 Literature review

Literature review in this research involved accepting 29 (23 for LLM based applications and design patterns + 6 regarding cognitive architectures' applicability in these applications) academic publications, and 20 blog posts were accepted. Half of these blog posts were rejected as they were biased towards promoting a singular company and/or denoted a niche set of results. Table 4.1 shows the keywords used for the database search and the total number of articles among which articles were chosen for analysis. This section displays the results identified from literature review.

Table 4.1: Database search results

Keywords	Material count
Generative AI architectures	7
LLM based applications	23
Generative AI applications	63
GenAI application	11
GenAI application architectures / design patterns	0
LLM based application architectures / design patterns	0
LLM	1600
cognitive architectures llm	383
cognitive architectures	57
ACT-R	1220
Soar cognitive architecture	116
ICARUS cognitive architecture	6

4.2.1 Observed design patterns

This subsection displays the results observed from a literature review on design patterns tailored to LLM based applications. Table 4.2 describes the different design patterns mentioned across various literature.

Table 4.2: Granular design Patterns for LLM based applications

S.No	Architecture	Description	Source
1	Retrieval-Augmented Generation (RAG) & Advanced RAG	Makes use of an external data source like a .pdf or a .csv file to add more context and improve the accuracy of the generated responses, using retrieval mechanisms.	[49], [50]
2	In-Context Learning	A few example inputs and outputs are provided to the model as context to improve the accuracy of the generated responses.	[51]
3	Ad-hoc	Implements a straightforward architecture using API calls to instantiate LLMs and prompt engineering is used to guide the model's output. This is a simpler structure without a sophisticated architecture.	[38]–[40], [52]
4	Multi-agent	Involves multiple agents that collaborate to solve complex tasks by breaking them down into simpler sub-tasks.	[4], [53], [54]
4.1	Multi-agent (Sequential)	The agents are organized in a sequential structure, where each agent passes outputs to the next agent.	[55]

4.2	Multi-agent (Cyclic)	The agents are organized in a cyclic structure where agents are executed in a loop, continuously refining tasks through feedback loops.	[55]
4.3	Multi-agent (Parallel)	Multiple agents work independently on different tasks simultaneously, synchronizing as needed.	[55]
4.4	Multi-agent (Master-Slave)	Features a master (manager) agent that directs the activities of subordinate (slave) agents, ensuring coordinated and synchronized task completion.	[56]
5	Usage of tools	Equips the LLM with tools for specific tasks like web searches or calculations, extending its capabilities.	[40], [57]–[59]
6	Chain-of-thought prompting	Breaks down complex problems into sequential steps where the LLM "thinks" with a flow of thought process and breaks down the tasks to perform them accurately.	[18], [40], [58]

4.2.2 Observed cognitive architectures

Having performed the search on cognitive architectures the result was four unique architectures. The chosen materials themselves apply cognitive architectures primarily in the role of robotics, however, they provide valuable insights into the functioning of the individual architectures. This research aided in understanding the breakdown of cognitive architectures and the different entities and functions associated with such an architecture.

1. **Soar:** According to Laird, Soar is a general cognitive architecture that provides components to create AI agents to mimic the approach found in humans. The major components involved in this architecture would be the Long-term and working memories, processing modules, and mechanisms to help with learning. The working memory is in charge of the agent's situational awareness. The component level interaction in the working memory contributes significantly to the agent behaviour.
   Furthermore, condition-action pairs in the Soar architecture are represented by production rules, which are essentially rules, which suggest the system to take an action during a state of the system.
   Another important part of the architecture is the decision cycle, where the system operates in a continuous cycle to select a production rule and apply the action associated with the rule to the current state. Finally, this knowledge is updated in the working memory. This is in turn useful while Soar updates its memory based on successful and failed actions [60].
2. **ACT-R:** According to Langley, Laird, and Rogers, ACT-R or Adaptive Con-

trol of Thought-Rational as an architecture has the primary goal of producing a psychologically motivated model. ACT-R 6 has the following modules each of which handles different types of information.

- Sensory module - Visual processing
- Motor module - Action
- Intentional module - Goals
- Declarative module - Long-term declarative knowledge

Each module has a buffer called a chunk, which holds a relational declarative structure forming ACT-R's short-term memory. On the other hand, ACT-R also has a long-term memory which comprises of production rules which, similar to Soar, coordinate different actions to different states [61].

According to Chong, Tan, and Ng, Since ACT-R takes inspiration primarily from the human brain, it is modelled in such a way that the various components of this architecture mimic human cognition. The goal buffer in ACT-R helps to keep track of the internal state during problem solving and declarative memory acts as the long-term memory [62].

3. **ICARUS:** According to Chong, Tan, and Ng, ICARUS primarily focuses on the development of physical agents. This integrates perception and action with cognition. By aligning sensory input directly with cognitive processes and motor responses, this architecture aims to produce agents that exhibit autonomy and adaptability in different settings. The primary architectural components are the perceptual buffer, the conceptual memory, the skill memory and the motor buffer. Conceptual and skill memories are further classified into long and short term memories [62].
4. **Subsumption:** According to Chong, Tan, and Ng, this architecture decomposes a problem in terms of robotic behaviour. This architecture is more ideal for reflexive tasks but would not be highly useful for cognitive functions. In contrast to the aforementioned architectures, this does not utilize a long and short term memory system but relies solely on sensors for the perception aspect [62].

By understanding the functioning of these architectures, it is clearer to identify different components and place them with existing LLM based application architectures. From the mentioned cognitive architectures, the common entities between the architectures are the presence of long and short term memories, perception of the environment as input, a target state or goal for the agent to achieve, a mechanism to solve the problem, reason and learn. These observed entities are high-level and exhibit minimal variations across different cognitive architectures. For example, according to Chong, Tan, and Ng, the perception for the Soar architecture as input is stored in the working memory while the perception for Subsumption is simply available through sensors but is not stored anywhere because of the lack of memories in the architecture [62].

To summarize, the following are individual high level entities identified across multiple cognitive architectures.

- **Perception:** Processes incoming sensory data from the environment.
- **Memory:** Stores information for future use (short-term and long-term).
- **Learning:** Helps the system learn from its surroundings and adapt to different

environments.
- **Reasoning:** Processes information to draw conclusions and make decisions.
- **Action Selection:** Chooses actions to achieve goals based on perception, memory, and reasoning.
- **Communication:** Enables agents to exchange information with each other (in multi-agent architectures).

Having gathered information on the operational mechanisms of various cognitive architectures, the relevance of these architectural components is investigated within LLM-based applications in below subsections.

4.3 Library of LLM design patterns

In this study, various LLM-dependent applications were analyzed to identify recurring architectural patterns. This section summarizes the general design patterns that emerged from the investigation, displayed in a GangOfFours style with the intent, motivation, structure, applicability, benefits, drawbacks, related patterns and known uses of the design patterns. Moreover, the design patterns are also classified based on if they are behavioural, structural or LLM interaction design patterns.

4.3.1 Multi-Agent (Role-based)

Classification: Behavioural

Intent: To manage tasks based on the roles assigned to the agents.

Motivation: This pattern promotes a clear division of responsibilities and specialization among agents, enhancing task efficiency and organization. Assigning specific roles allows each agent to focus on its designated function, leading to better overall system performance and scalability.

Structure: The following is the structure of this design pattern.
1. **Assigning roles:** Each agent is assigned a role that mirrors their real-life counterpart, enabling the agent to effectively emulate the person's behavior and actions.
2. **Interaction:** Agents can interact with each other depending on their roles. This interaction can lead to cyclic, hierarchical, or sequential behaviour between agents.
3. **Optimization:** Agents are optimized using tools and functions, based on the requirements for their roles.

Applicability:
1. **Pro-indications:** When the clear division of tasks between agents is required.
2. **Contra-indications:** When tasks do not require split of roles or cannot be separated into different roles.

Benefits:
1. Task management and coordination between agents are made more comprehensible.

2. Optimization of an agent to a specialized role would enhance efficiency.

Drawbacks:

1. Since this follows a human-like approach, it would be difficult to create and maintain roles at a larger scale.
2. If roles higher in the hierarchy are not delegated appropriately, it could lead to bottlenecks.

Related Patterns: Multi-Agent (Hierarchical), Multi-Agent (Sequential)

Known Uses: MetaGPT, CrewAI, AutoGen.

4.3.2 Multi-Agent (Hierarchical)

Classification: Behavioural

Intent: Worker and manager LLM agents work in sync to complete an input task.

Motivation: This pattern promotes a clear division of responsibilities and specialization among worker agents. Assigning a manager agent to delegate tasks allows the organization to focus on its input task, leading to better overall system performance and scalability, as it mimics a real-life SDLC.

Structure: The following is the structure of this design pattern.

1. **Manager agent:** A manager agent is assigned to delegate tasks intelligently to employee or worker agents. The manager agent is also responsible for reviewing the output generated by the other agents to display to the end user.
2. **Worker agents:** Specialized agents that are responsible for a smaller portion of the task.

Applicability:

1. **Pro-indications:** When the output accuracy is favoured.
2. **Contra-indications:** When tasks do not require split of roles or cannot be separated into different roles.

Benefits:

1. Task management and coordination between agents are made more comprehensible.
2. Optimization of an agent to a specialized role would enhance efficiency of individual agents.
3. The manager serving as a reviewer agent aims to improve the accuracy of the generated output.

Drawbacks:

1. A singular manager agent can face issues with bottlenecking when managing multiple worker agents.

Related Patterns: Multi-Agent (Role-based), Multi-Agent (Sequential)

Known Uses: MetaGPT, CrewAI.

4.3.3 Multi-Agent (Sequential)

Classification: Behavioural

Intent: To manage agents performing tasks in a sequence, each passing outputs to the next agent.

Motivation: Sequential task execution ensures orderly progression and dependency management.

Structure: The following is the structure of this design pattern.

1. **Arrangement:** Agents are arranged in a sequence so one agent receives the output of another, till the task is completed.

Applicability:

1. **Pro-indications:** When tasks need to be completed in a specific order.
2. **Contra-indications:** When tasks do not have the orderly requirement and can be completed in parallel.

Benefits:

1. Progress of tasks takes place in an orderly fashion.
2. There is less complication in dependency management.

Drawbacks:

1. A bottleneck agent can slow down the progression of the entire flow.

Related Patterns: Multi-Agent (Cyclic), Multi-Agent (Parallel)
Known Uses: GPTPilot

4.3.4 Multi-Agent (Cyclic)

Classification: Behavioural

Intent: To continuously refine tasks through feedback loops among agents.

Motivation: Continuous feedback loops through cyclic interactions can improve accuracy.

Structure: The following is the structure of this design pattern.

1. **Actor:** An actor agent performs the task assigned to it.
2. **Observer:** A feedback agent constantly provides feedback to the actor in a loop, to increase the quality of output.

Applicability:

1. **Pro-indications:** When tasks need iterative refinement and high accuracy.
2. **Contra-indications:** When tasks are simple and linear.

Benefits:

1. Refinement of the output continuously leads to increased accuracy.

Drawbacks:

1. Iterative reasoning can be a slow process.
2. When used for a simple task, it could consume more resources than required.

Related Patterns: Multi-Agent (Sequential), Multi-Agent (Parallel)
Known Uses: DroidAgent

4.3.5 Multi-Agent (Parallel)

Classification: Structural

Intent: To execute multiple tasks simultaneously by making agents work in parallel on different tasks.

Motivation: Parallel processing increases efficiency and reduces execution time due to concurrency.

Structure: The following is the structure of this design pattern.

1. **Synchronization:** Agents would work independently on several tasks but would reconvene and synchronize should it be needed.

Applicability:

1. **Pro-indications:** When tasks can afford to be simulated in parallel.
2. **Contra-indications:** When tasks are highly dependent of each other and cannot be executed in parallel.

Benefits:

1. Efficiency and reduced time consumption while performing tasks concurrently.

Drawbacks:

1. Independent tasks would be difficult to synchronize.

Related Patterns: Multi-Agent (Sequential), Multi-Agent (Cyclic)
Known Uses: DroidAgent

4.3.6 Retrieval-Augmented Generation (RAG)

Classification: LLM Interaction
Intent: To enhance LLM functionality by making use of external data retrieval techniques.

Motivation: This can help make use of external data as context, to improve the relevance of the LLM generated content.

Structure: The following is the structure of this design pattern.

1. **Embedding Model:** A neural network model or an API call, which embeds input data (text, picture or video) into vector form.
2. **Vector Database:** The storage medium for storing the embedded vectors.
3. **Retrieval:** Retrieval mechanism include different techniques to address similarity. Metrics like Manhattan, Cosine-similarity etc, can be used to retrieve relevant data from the Vector Store.
4. **Data Filter (Optional):** Data filter is used to restrict the free outflow of confidential data.

Applicability:

1. **Pro-indications:** When the LLM needs to provide accurate information with an extended context.
2. **Contra-indications:** When external context is unnecessary for the use-case.

Benefits:

1. Enhances the accuracy of the output responses.

Drawbacks:

1. Additional complexity due to retrieval mechanisms.
2. Complexity in embedding relevant data and choosing the right vector store for the use-case.

Related Patterns: In-Context Learning, Prompt engineering

Known Uses: Continue, DroidAgent

4.3.7 Usage of tools

Classification: LLM interaction (since it enhances the LLMs)

Intent: To enhance LLM functionality by equipping it with various tools to access the web, access GitHub etc.

Motivation: The usability of LLMs are enhanced by equipping them with higher capabilities and functionalities.

Structure: The following is the structure of this design pattern.

1. **Function calling:** The API calls to the LLMs can have an additional parameter of a function definition, where the LLM can intelligently choose to make use of the function to complete a task.

Applicability:

1. **Pro-indications:** When the LLM needs to perform specialized tasks that are better handled by dedicated tools. For example, user requires an LLM to get and describe weather data of a certain place. Provided a function which say, calls the weather API and retrieves the data, the LLM can make use of this function to interpret said data.
2. **Contra-indications:** When tools perform redundant tasks, which can be accomplished without them.

Benefits:

1. Expands the capability scope of LLMs.
2. Can improve the accuracy of relevant responses.
3. Usage of several tools, as required by the use-case.

Drawbacks:

1. External dependency of third part APIs can cause latency and even availability issues.

Related Patterns: In-Context Learning, Pipeline, Chain-of-Thought Prompting

Known Uses: DroidAgent, Continue

4.3.8 Action Planner

Classification: LLM Interaction
Intent: To create a structured sequence of actions or sub-tasks based on the input task or goal. This could guide the LLM better to create relevant agents or how to go about each task.

Motivation: Ensures that complex tasks are broken down into manageable steps, hence providing better clarity for the LLM. By generating an action plan, the LLM can systematically address each aspect of the task, leading to more accurate outcomes.

Structure: The following is the structure of this design pattern.

1. **Task Analysis:** The LLM analyses the input task to classify components as needed and the required actions.
2. **Sequential planning:** The LLM generates the sequence of actions to be taken into account to accomplish the task.
3. **Execution steps:** The LLM generates detailed instructions to be executed at each step.

Applicability:

1. **Pro-indications:** When tasks are complex and require a clear, structured approach.
2. **Contra-indications:** When a flexible, adaptive approach is more suitable than a fixed plan.

Benefits:

1. Provides modularity to the topic by breaking down complex tasks into simpler steps.
2. Enhances the ability of the LLMs to generate coherent outputs.

Drawbacks:

1. Complexity in Prompt Engineering.
2. Requires additional design time to ensure that each step in the plan is relevant.

Related Patterns: Chain-of-Thought Prompting, Usage of Tools, Task Decomposition.
Known Uses: DroidAgent, GPTPilot

4.3.9 Chain-of-thought prompting

Classification: LLM Interaction
Intent: To improve the response accuracy by breaking down tasks into sequential steps, guiding the thought process of the LLM step-by-step. This is different from Action Planning in a sense that Action Planner generates a set of actions for the LLM to do, while Chain-of-thought guides the LLM through it's thought process. This can be seen in CSV_Agents by Langchain [40].

Motivation: This approach ensures clarity and precision in handling complex tasks by decomposing them into smaller steps.

Structure: The following is the structure of this design pattern.

1. **Decomposition:** Here the input task is broken down step-by-step, where each thought in-between builds on the previous thought to guide the LLM's response.

Applicability:

1. **Pro-indications:** When tasks are complex and benefit from being broken down into simpler components.
2. **Contra-indications:** When a task is simple and do not require decomposition.

Benefits:

1. Logical thought progression of the LLM helps it to generate accurate responses.
2. Solves a problem by divide and conquer.

Drawbacks:

1. Sequential nature of processing can cause slowdowns.

Related Patterns: In-Context Learning, Usage of Tools

Known Uses: MetaGPT, Continue, GPTPilot, LangChain

4.3.10 In-context learning

Classification: Structural

Intent: To provide LLMs with examples and contextual information within the input prompt to guide response generation. This is more of a variation of prompt engineering.

Motivation: This approach enriches the LLM's adaptability and helps it to understand specific contexts, thus making it optimised for those contexts.

Structure: The following is the structure of this design pattern.

1. **Few-shot learning:** Few example contexts / scenarios are embedded within the input prompt to make the LLM understand the context of the provided task.

Applicability:

1. **Pro-indications:** When the LLM needs to generate responses based on a few example contexts.
2. **Contra-indications:** When additional context is not needed for a specific question.

Benefits:

1. Enhances the relevance and accuracy of LLM outputs.
2. LLMs could adapt to newer tasks based on the updated contexts.

Drawbacks:

1. Contextual examples need to be crafted carefully.

Related Patterns: Retrieval-Augmented Generation, Prompt Engineering

Known Uses: Continue, InvokeAI, GPTPilot

4.3.11 Pipeline

Classification: Behavioural
Intent: To process user inputs sequentially through various stages to generate the final output. Differs from Multi-agent (Sequential) in a way that it does not have agents operating sequentially, but rather a set of static and LLM based tasks.

Motivation: Sequential processing of user input ensures a structured approach to handle complex tasks in smaller stages.

Structure: The following is the structure of this design pattern.
1. **Pipeline:** Inputs are processed through a defined set of stages, where each stage adds value that impacts the final output.

Applicability:
1. **Pro-indications:** When tasks need to be processed in a structured sequence.
2. **Contra-indications:** When tasks can be processed independently or in parallel.

Benefits:
1. Ensures orderly progression of tasks.
2. Furthermore, simplifies the processing stage of complex tasks.

Drawbacks:
1. Sequential processing can lead to bottlenecks, which can cause delays.

Related Patterns: Multi-Agent (Sequential Interaction)
Known Uses: InvokeAI

4.3.12 Prompt Engineering

Classification: Structural
Intent: To interact with LLM APIs directly without a predefined rigid structure, offering flexibility and simplicity.

Motivation: Prompting to an LLM via an API can save a lot of time while prototyping.

Structure: The following is the structure of this design pattern.
1. **Zero-shot learning:** The architecture involves direct API calls to LLM services without a fixed structure, simply manipulating prompts for a desired output.

Applicability:
1. **Pro-indications:** When faster prototyping is required.
2. **Contra-indications:** When a structured architecture is needed for maintenance.

Benefits:
1. Highly simplifies the integration of LLMs to an application.

Drawbacks:
1. Would not scale well for complex use-cases.

Table 4.3: Granularity levels of the design patterns.

S.No	Design pattern	Granularity level
1	Multi-Agent (Role-based)	Architectural
2	Multi-Agent (Hierarchical)	Architectural
3	Multi-Agent (Sequential Interaction)	Architectural
4	Multi-Agent (Cyclic Interaction)	Architectural
5	Multi-Agent (Parallel Processing)	Architectural
6	Retrieval-Augmented Generation	Mid-level
7	In-Context Learning	Idiom
8	Prompt Engineering	Idiom
9	Pipeline	Architectural
10	Usage of tools	Idiom
11	Chain-of-thought prompting	Mid-level
12	Action Planner	Idiom

Related Patterns: Pipeline, In-context learning
Known Uses: Continue, InvokeAI, BabyAGI

4.3.13 Summary

The above subsections mention the individual patterns we were able to identify among the 9 different projects. It is to be noted that although they are portrayed as individual patterns, they can be combined to a larger pattern based on the use-case requirement. The architectures can be classified into high, low or mid-level based on their position in the application. For example, a multi-agent architecture's agent can be built using RAG (which inturn has a step where it uses Prompt Engineering), which makes multi-agent a high level architecture and RAG a mid-level architecture. A pattern which is the closest to an LLM, as in through APIs or locally, is considered as low-level architecture. Architectural level depicts high-level architecture while idiom depicts a low-level architecture. Table 4.3 depicts the different granularity levels of different observed design patterns.

4.4 Design patterns in LLM based applications

The following section is split based on the results of 9 different projects that were subject to analysis. In Table 4.4, the projects are categorized based on their architectures and the types of output they generate, as indicated in the *Output Type* column. This categorization helps in understanding the diverse applications and functionalities of LLM-dependent systems, highlighting the versatility and adaptability of these architectures. Moreover, the results deduced from the analysis of the projects are presented in the format,

- Title
  - Components
  - Flow Structure

- Design pattern(s)
- Comments

The selected projects were built to scale. It was in the criteria to choose "large enough" projects. Most of these projects made use of a Multi-agent architecture in the higher level. Upon a deep exploration, it was observed that different use-cases required different unique patterns in said Multi-agent architecture. Patterns were observed on how the agents interacted with each other and on how the agents themselves were built.

There were two projects that were excluded due to the interviews not yielding any useful information, namely GPT-Pilot and PR-Agent. The interviews can be found in A.2.2 in the appendix. GPT-Pilot did not have any architectural, structural, or central concept documentation, and the developers did not seem to have any internal documentation of such either. PR-Agent had some documentation, but nothing substantial. There were also three projects that were excluded because the developers did not respond in any way. These were Autolabel, AutoGPT, and GPT4All. One project, RasaGPT, was excluded because there was no practical public way to contact the developer and because the documentation could only be accessed through running the program. And all attempts at running the program failed, meaning the documents were inaccessible.

Table 4.4: Selected projects, architectures, output.

S.No	Project Name	Architecture	Output type
1	MetaGPT	Multi-agent (Role-based) + Assembly Line	Hybrid
2	DroidAgent	Multi-agent (Cyclic) + Tools + Action Planner	Code-generative
3	Promptify	Static (Outlier)	Prompt-generative (Outlier)
4	Continue	In-Context + RAG + Ad-hoc	Code-generative
5	InvokeAI	In-context + Calls to API	Image-generative
6	BabyAGI	Multi-agent (Cyclic) + Calls to API + RAG	Hybrid
7	AutoGen	Multi-agent (Role-based) + RAG	Hybrid
8	GPTPilot	Multi-agent (Sequential) + In-context	Code-generative
9	CrewAI	Multi-agent (Role-based)	Hybrid

4.4.1 MetaGPT

MetaGPT is a framework which aids in encoding human Standard Operating Procedures (SOP) to LLM agents, as mentioned by Hong, Zhuge, Chen, *et al.* [4]. This project offers a multi-agent architecture style where each agent has a role defined by the user, along with constraints to the mentioned role. A workflow is established by defining the agent roles and skillset for each role.

Components

The major components from MetaGPT are the individual agents responsible for the different roles in the SDLC. The file `software_company.py` defines the core logic of the virtual software company using LLM based roles. Furthermore, the `roles/` directory contains the classes for different roles. The following are the roles in the directory.

1. Architect
2. Assistant
3. Customer Service
4. Engineer
5. Invoice OCR Assistant
6. Product Manager
7. Project Manager
8. QA Engineer
9. Researcher
10. Searcher
11. Sales
12. SK Agent
13. Teacher
14. Tutorial Assistant

Flow structure

The flow in MetaGPT is similar to that of a software company. Thus, the different agents (GPT based roles) collaboratively simulate the company. MetaGPT requires a one-line SOP query, which is decomposed appropriately and the tasks are divided among agents, which is inturn used to generate user stories, competitive analysis, requirements, APIs and documentation. The core roles in MetaGPT are mentioned in the previous subsection. The project offers a highly customizable framework, with multiple supported LLM providers, detailed setup options, thus allowing it to be suitable for different requirements and contexts.

Design patterns

MetaGPT makes use of a role-based multi-agent system where the roles are predefined. Moreover, it follows an assembly line architecture so the query goes through multiple steps in between with multiple iterations of refinement before the actual product is made visible to the end user.

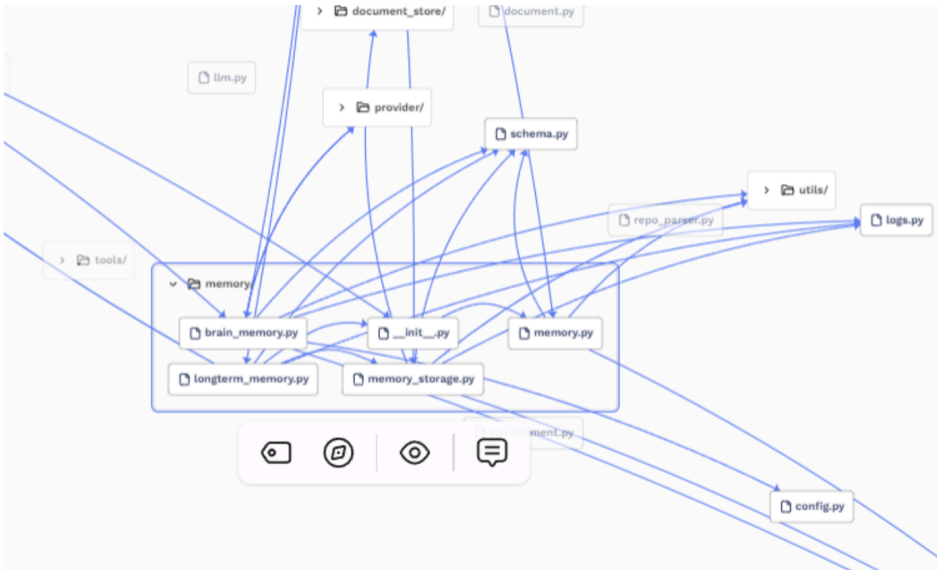

Figure 4.1: Dependency graph for MetaGPT

Comments

The initial method of analysing MetaGPT, through visual analysis of the dependency map drawn by CodeSee, did not yield any particularly useful results. The code itself was not structured in a helpful manner as shown in Figure 4.1, as the dependencies were very unorganised, to the point where you could not tell if there was any planned structure besides the fact that files were divided amongst different folders. The one insight gained from this, however, was that there was extensive use of inheritance polymorphism in several parts of the codebase. Unfortunately, since CodeSee is now defunct, no visual examples of the dependency map's visuals are available.

Following this, the developers were asked about the internal structure of MetaGPT in its official Discord server, which did not yield descriptive responses. The responses that were given were instead two links to the documentation of MetaGPT. This did, at least to a certain extent, show how the application worked internally in a more abstract manner. These two links contained two very useful graphics, as shown in figure **??**. These graphics gave insight as to how LLM multi-agent frameworks can be built, at least abstractly.

Since MetaGPT was the first project to be analysed, no process had been established at that point during the thesis project. Hence, the methodology and results differ from the projects analysed later. MetaGPT also took longer to analyse than the other projects since, due to the lack of an established standard procedure, a lot of time was spent in the visual analysis phase. This phase also involved a research

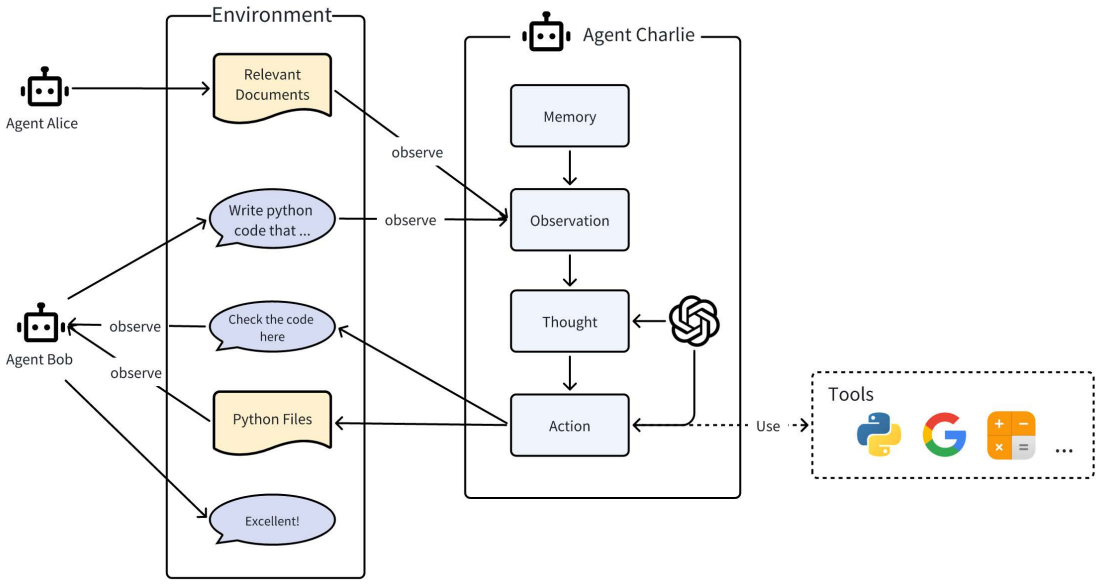

Figure 4.2: MetaGPT's agent framework [63].

analysis of MetaGPT which revealed that the project made use of a multi-agent architecture. Finally, the interactions of the agents in the framework was analysed and it was assumed to mimic a process called the assembly line paradigm, where according to Rekiek and Delchambre, individual parts of a product are created and assembled down the road [64]. This enables the interacting agents to form loops or a cycle. An example provided the documentation can aid better in understanding the workflow. Consider a scenario with three agents and different roles: a programmer, a tester, and a reviewer. This is a common SOP in the current world. The programmer agent writes the main code while the tester agent writes a test suite for the main code. Finally, the reviewer agent acts as a quality checker for the coverage and the test suites. Each agent observes the output from the upstream agent and publishes its own output for the downstream agent. This forms a cycle of interactions among the agents [65].

4.4.2 DroidAgent

According to Yoon, Feldt, and Yoo, DroidAgent is an intent driven mobile GUI testing framework which makes use of LLM agents. It is built as a modular framework of multiple LLM instances. It sets relevant task goals and tries to achieve them by interacting with the application [10].

Components

Upon analysis of DroidAgent, it was perceived that it was also built using a multi-agent architecture with four LLM-based agents and three different memory modules. This project makes use of a Multi-agent architecture, with four primary agents and three different memory modules. The breakdown of each individual component is

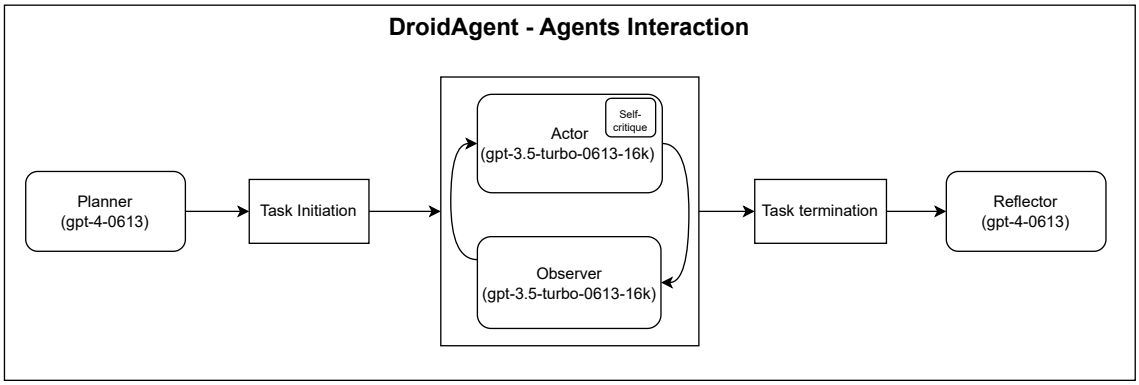

Figure 4.3: Agents interaction in DroidAgent [10].

as follows.

- Agents
  - Task Planner
  - Actor
  - Observer
  - Reflector
- Loops
  - Outer loop, the full program execution loop.
  - Inner loop, the interaction pattern between the Actor and Observer agents.
- Memory
  - Working memory
  - Task history, or long term, memory. Which stores the previous tasks performed, regardless of whether they succeeded or failed, and the background information such as the ultimate goal.
  - Widget knowledge, or spatial, memory. Which stores GUI state layout and information, not just the current and previous GUI states.

Flow structure

The agents interact with each other in a cyclic manner, followed by a linear flow of results. Figure 4.3 displays how the agents interact between each other in the application.

Design patterns

The observed design patterns from DroidAgent are a Multi-agent architecture where the agents interact in a cyclic manner. This was visible especially in the formation of the inner and outer loop and the "self-critique" function of DroidAgent. Moreover, we were also able to observe the usage of tools in the project. Finally, the Action Planner contributed as another unique design pattern by itself.

Comments

DroidAgent was also observed to use different LLMs for different agents. The developers did not require the power of GPT-4 for all the tasks, so they used GPT-3.5 for some agents [10]. DroidAgent's structure uses a static multi-agent framework.

4.4.3 Promptify

Promptify was a lot simpler and smaller project compared to MetaGPT and DroidAgent. Promptify had one example which was very simple, the entire file being only 15 lines long as seen in Figure 4.5.

Components

Promptify requires 4 objects to function, and the integral being the prompter and pipeline. These two classes, specifically their major functions, had very good descriptions of their functionality. This made it possible draw the structure of the application quickly, leading to the chart as seen in Figure 4.6.

Flow structure

The flow structure of Promptify is as follows:

1. User Input - Text.
2. The **pipeline** sends the input to the prompter.
3. The **prompter** reformats the input into a finished prompt using a preselected format template and returns it to the pipeline. The template is of a special file type, ".jinja", which is a file type associated with the Jinja template engine [66].
4. The pipeline then sends the finished prompt to the chosen LLM.
5. The **LLM** gives a response.
6. The pipeline sends the response as output to a logger
7. The pipeline sends the output to the user.

First the pipeline sends the input to the prompter. The prompter then reformats the input into a finished prompt using a preselected format template. The template is of a special file type. .jinja to be precise, which is 8a file type associated with the Jinja template engine [66]. The pipeline then sends the finished prompt to the chosen LLM, which then gives a response which is then sent by the pipeline to a logger and as final output to the user.

Design patterns

Considering the nature of the application being static, there were no observable design patterns except the usage of an API call to the LLM.

Comments

Promptify is one of the outlier projects that were analysed, since it sparked an interesting discussion about the need of Prompt Engineering as a design pattern in

Quick tour

To immediately use a LLM model for your NLP task, we provide the `Pipeline` API.

```
from promptify import Prompter,OpenAI, Pipeline

sentence     =  """The patient is a 93-year-old female with a medical
                history of chronic right hip pain, osteoporosis,
                hypertension, depression, and chronic atrial
                fibrillation admitted for evaluation and management
                of severe nausea and vomiting and urinary tract
                infection"""

model        = OpenAI(api_key) # or `HubModel()` for Huggingface-based inference or 'Azure' etc
prompter     = Prompter('ner.jinja') # select a template or provide custom template
pipe         = Pipeline(prompter , model)


result = pipe.fit(sentence, domain="medical", labels=None)


### Output

[
    {"E": "93-year-old", "T": "Age"},
    {"E": "chronic right hip pain", "T": "Medical Condition"},
    {"E": "osteoporosis", "T": "Medical Condition"},
    {"E": "hypertension", "T": "Medical Condition"},
    {"E": "depression", "T": "Medical Condition"},
    {"E": "chronic atrial fibrillation", "T": "Medical Condition"},
    {"E": "severe nausea and vomiting", "T": "Symptom"},
    {"E": "urinary tract infection", "T": "Medical Condition"},
    {"Branch": "Internal Medicine", "Group": "Geriatrics"},
]
```

Figure 4.4: Promptify sample code

Code Blame 15 lines (12 loc) · 736 Bytes

```
1   from promptify import Prompter,OpenAI, Pipeline
2
3   sentence     =  "The patient is a 93-year-old female with a medical
4                   history of chronic right hip pain, osteoporosis,
5                   hypertension, depression, and chronic atrial
6                   fibrillation admitted for evaluation and management
7                   of severe nausea and vomiting and urinary tract
8                   infection"
9
10  model        = OpenAI(api_key) # or `HubModel()` for Huggingface-based inference or 'Azure' etc
11  prompter     = Prompter('ner.jinja') # select a template or provide custom template
12  pipe         = Pipeline(prompter , model)
13
14  output = pipe.fit(text_input=sentence, domain="medical", labels=None)
15  print(output)
```

Figure 4.5: Promptify only file in the examples directory [67].

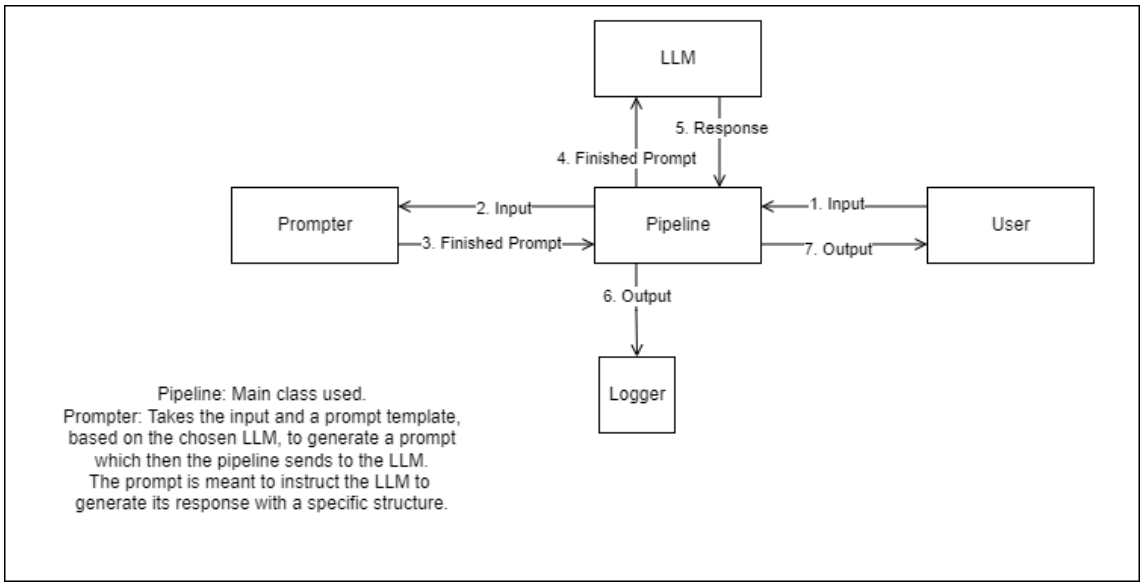

Figure 4.6: Extracted structure of Promptify

certain applications.

4.4.4 Continue

Continue is an autopilot IDE extension, which aids a developer to understand code, write comments, generate test cases for highlighted code using modern LLMs.

Components

The major components as seen in the Continue source code are as follows:

- Serializer
- Context Providers
- Chat Handler
- Pruner
- Formatter
- LLM Adapter

Upon further analysis, it was observed that Continue makes use of a typescript architecture (Core, data and DI) than a LLM specific architecture. For LLM related tasks, Continue makes use of a combination of in-context learning and API calls. The usage of API calls and the presence of in-context learning is identified by analysing the code, since there is no explicit mention of it in the documentation. Furthermore, the project also makes use of RAG to enhance the LLM inputs with relevant data. As mentioned above, there was a lack of an official documentation about the project structure but rather just the usage of the extension in Visual Studio Code.

Flow structure

Continue's components are as follows:

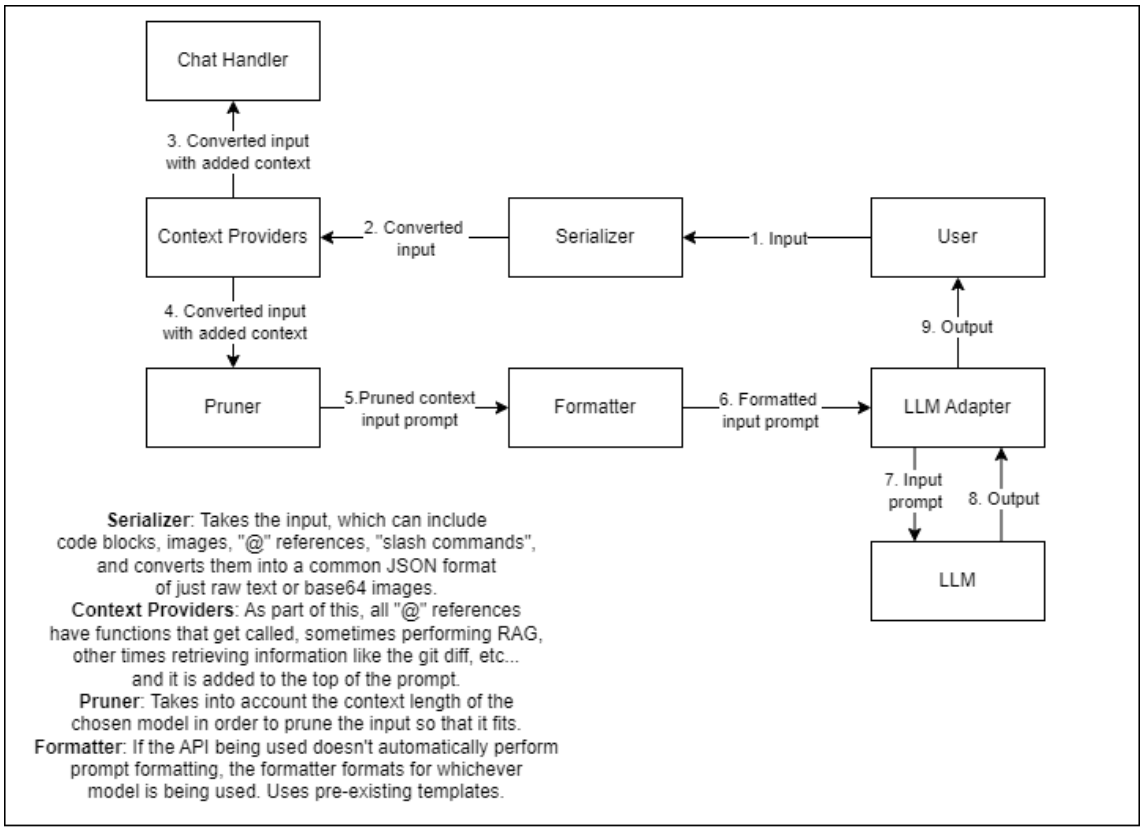

Figure 4.7: Extracted structure of Continue based on the developer's description.

1. **Serializer**: Converts multi-media input into a common JSON format of just raw text or base64 images (A.1.1).
2. **Context Providers**: Can be seen as either its own component or as part of the Serializer. All "@" references, being one type of input that the application can process, have functions that get called. These functions include, but are not limited to, those that perform RAG or retrieval information like the git diff. The results from the functions are then added to the prompt as extra context (A.1.2).
3. **Chat Handler**: Adds the prompt to all of the previous messages from the conversation (A.1.3).
4. **Pruner**: Takes into account the context length of the model in order to prune the input so that it fits (A.1.4).
5. **Formatter**: If the API being used doesn't automatically do prompt formatting, it formats the prompt for whichever model is being used (A.1.5).
6. **LLM Adapter**: Sends the prompt to the LLM API (A.1.6). This is the abstraction that lets the application send prompts to any LLM API.
7. Frontend UI.

Figure 4.7 depicts a high level flow of the structure of Continue.

Design patterns

Continue primarily made use of In-Context learning and Retrieval Augmented Generation for its code generation functions. There is also a use of API calls to actually send and receive queries to the LLM instances hosted online. There are also a case to be made that Continue uses some kind of, or at least something adjacent to an, adapter pattern in the LLM Adapter component, since its stated goal is to allow any LLM to be used.

Comments

Continue's codebase structure did not lend itself well to static code analysis as it wasn't structured in a module or entity fashion. This is further corroborated by the developers own words when asked about it A.2.1.1. This means that the components are not formal implemented structures but informal abstracted structures. There was also no real documentation describing the structure of the codebase nor any example files. Contacting the developers over their public Discord server was very fruitful, as they were very helpful people who seemed happy to help. The questions, which were the precursor to the standard questions, that was asked of them yielded the response found in A.2.1.1, with the sources used in 4.4.4 being links that were in the response. These were also used in the description of the components. The phrasing of "...so that it fits" in step 4 may be a bit vague, but considering that the "Pruner" is in the "countTokens" class, it's reasonable to assume that he means that it prunes the input as to not consume too many tokens.

4.4.5 InvokeAI

InvokeAI is described as a creative engine for diffusion models, meaning it generates images using generative AI models such as Stable Diffusion. Attempts at contacting proper developers of InvokeAI via their official discord server were not successful, however, a contributor did offer aid using which we were able to deduce the components and flow of the project.

Components

The InvokeAI structure was divided into the following components:

- **Frontend:** React and Typescript.
- **Backend:** Python for AI (pytorch library) and a node-based architecture, where modular processing steps are linked together in a graph to form the full pipeline.

Flow structure

According to the contributor, the following steps are how InvokeAI functioned.

- When you invoke a basic inference pipline from the frontend it builds a graph from a pre-selected format based on the inputs.

- The initial values in the graph are filled from the settings in the web UI, and some nodes in the graph may be inserted/removed based on what settings are enabled.
- The graph is sent to the backend API where it is put in a queue behind any previous or ongoing processes.
- Once the graph reaches execution, it calls the '`invoke()`' method of each node and passes the output values to the input values of the next node in the sequence.
- Usually the last stage is to have a node that saves an image output from one of the previous nodes, but not always.

Additionally, the contributor also added the following considerations:

- Rather than using a pre-selected graph format, there is also a Workflow Editor that allows users to structure their own node graph for better control. For examples of what those look like, check out share-your-workflows.
- Outputs from nodes are stored in a database or to disk, and caching is enabled such that if a node is executed in multiple graphs with all of the same inputs, it will skip the processing and use the cached result instead.
- If a graph fails from some internal error, then the local process stops and moves on to the next graph in the queue."

A graphic of how the application works was provided by the contributor, which is shown in Figure 4.8.

Design patterns

InvokeAI primarily used in-context learning and calls to the Stable Diffusion API to generate results.

Comments

InvokeAI primarily makes use of react-typescript's component based architecture. Since the primary concern of the research is with the LLM aspect of the project, the `/backend` directory of the application was analysed. Here, it was observed that the project makes use of API calls to HuggingFace. The API calls were made to the `diffusers` package to access the stable diffusion models. Since InvokeAI requires an input prompt to generate images, it makes use of the current context hence also making use of In-context learning. It can also be argued that the multiple node arrangement in InvokeAI is similar to that of a multi-agent system, where each node can be an agent. However, it is more of a similarity and has not been explicitly stated anywhere. Hence it can be concluded that InvokeAI makes use of an In-context + ad-hoc architecture.

4.4.6 BabyAGI

BabyAGI is described as a task management system [68]. The wording used for it, however, describes it as a "script". But since the structure is very much a static multi-agent framework, making its structure similar enough to other projects reported in this thesis, it's included here regardless.

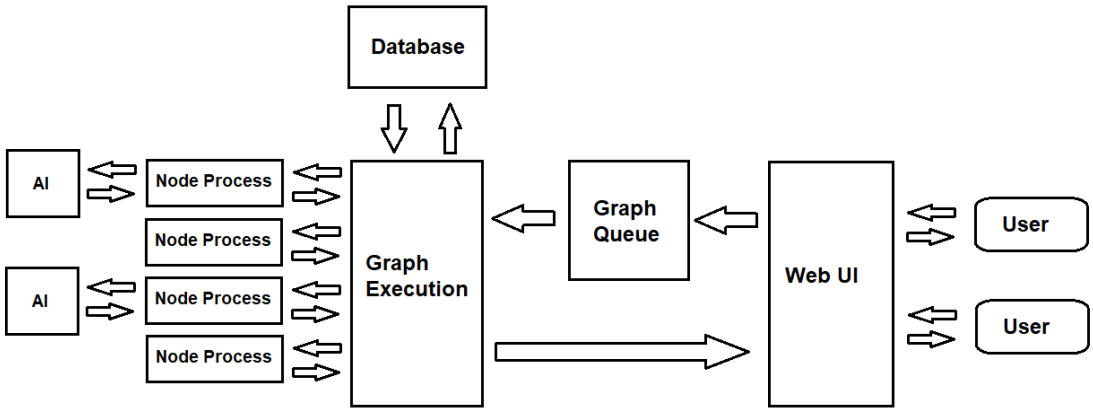

Figure 4.8: Structure of InvokeAI provided by a contributor.

Components

The different agents in the standard configuration shown and described on Github are the following [68]:

- **Execution agent**: This agent uses the OpenAI API to execute the tasks given to it. It takes two parameters: the task and its objective. It then uses the two parameters two construct a prompt, which it sends to the OpenAI API. The prompt also includes a description of the task. The execution agent then receives the output and returns it as a string.
- **Creation agent**: This agent uses the OpenAI API to creature new tasks based on the result and the stated objective of the previous task. Like the execution agent, the creation agent uses parameters to construct a prompt. However, unlike the execution agent, it requires four parameters: the current task list, the result of the previous task, the task description and the objective. Also unlike the execution agent, the string output, which is a list of new tasks, from the OpenAPI isn't returned as a string but as a list of dictionaries.
- **Prioritization agent**: This agent is more simple as it only takes the ID of the current task as its parameter and returns the newly prioritised task list as a numbered list.
- **Context agent**: The documentation doesn't actually state what the context agent does, but since when the documentation talks about Chroma or Weaviate, two vector database services [69], [70], it also talks about retrieving context which is the job of the context agent. This means that the context agent uses Chroma or Weaviate to store and retrieve task results for future context, and possible additional metadata.

Flow structure

BabyAGI had a satisfactory graphic of its flow structure on the main page of the GitHub repository, the one seen in 4.9. The graphic shows three steps whilst the description says four [68]. Although it seems like the first two described steps are part of the first illustrated step. The flow is described in the following steps:

1. Pull the first task from the task list and send it to the execution agent for task execution.
2. Enrich the result using the context agent and store the result in a vector database, Chroma or Weaviate to be precise.
3. Create new tasks, add them to the task list, and re-prioritise the tasks in the task list based on the result and objective of the previous task.

Design patterns

BabyAGI follows a multi-agent architectural style with each individual agent operating using calls to the OpenAI API. Since it is primarily multi-agent it is sufficient to observe the cycle formed by agents. Moreover, BabyAGI also makes use of context retrieval mechanisms indicating the usage of a RAG architecture with the Context Agent. The main file called `babycoder.py` consists of options to either use GPT-3.5-Turbo or GPT-4, and requires the user's own API key. Thus the cost of the architecture depends on the choice of model.

Comments

There is a total of twelve agents in the main file ready to use, despite it not being mentioned in the Github description. The agents are programmed by simple prompt engineering techniques. It is safe to conclude that the agents interact with each other in a pipeline or a cycle and are programmed by ad-hoc techniques.

4.4.7 AutoGen

AutoGen is a multi-agent framework, offering highly customisable agents able to converse with both each other and with human users. The agents are also able to use tools, such as code executing ones.

Components

AutoGen, like MetaGPT, does show how its agents are built. It is however very simplistic, only showing what built-in components are available for the example agent, as shown in Figure 4.10. The figure also shows that the standard agent type is open for extension with the use of custom components. Judging by what is written in the tutorial [71], AutoGen's agents are very modular as the built-in components can be turned on and off to suit the user's need, along with plugging in other components to further customise the agent.

Flow structure

AutoGen has three built-in conversational patterns as environments for the agents to interact in, along with the ability to use custom, user made, conversational patterns [72]. This means that the flow structure is highly customisable. The four built-in conversational patterns are:

1. **Two-Agent Chat:** This is the simplest of the three patterns. The chat is initiated with an initial message to one of the agents along with the context.

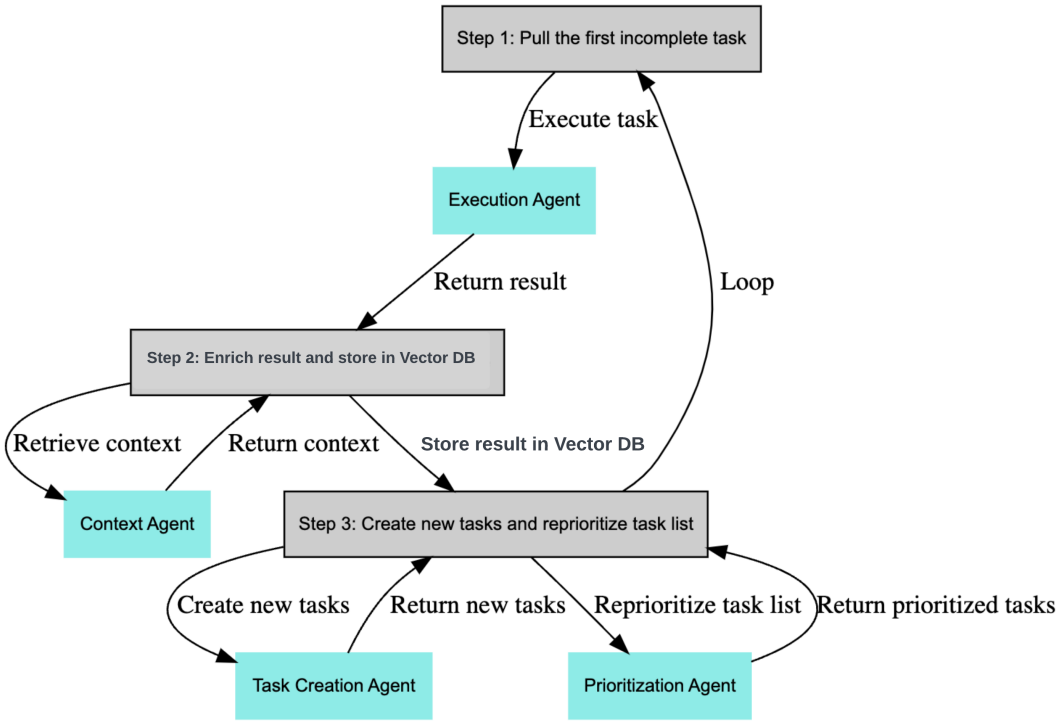

Figure 4.9: Structure of BabyAGI found on its GitHub page [68].

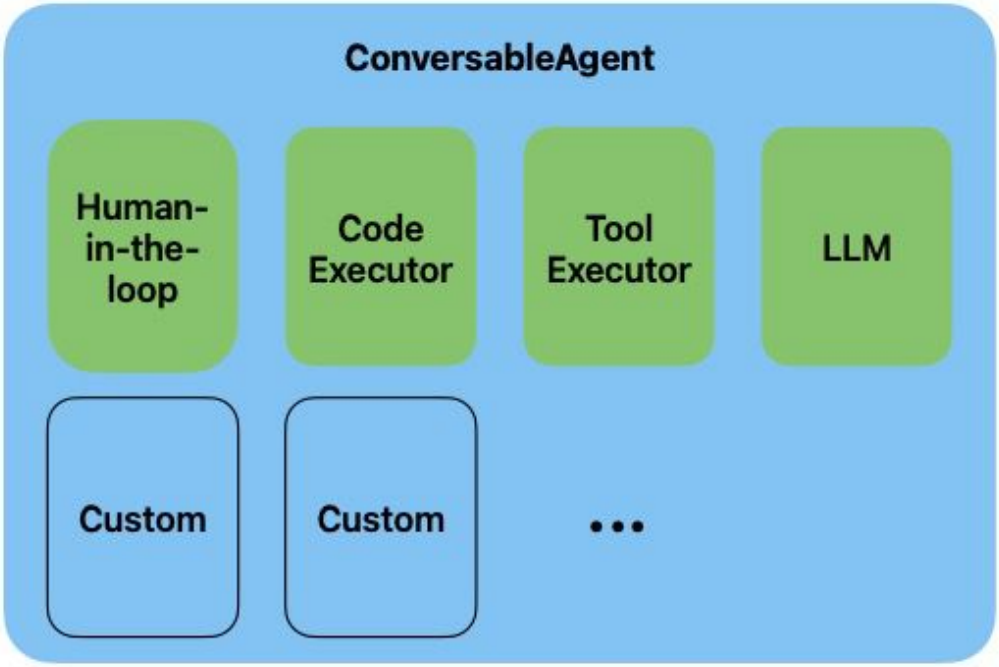

Figure 4.10: AutoGen agent example [71].

The two agents then proceed to exchange messages until the chat ends. The chat result is then summarised and sent to the user as output. This is shown in Figure 4.11.

2. **Sequential Chats:** This pattern is a string of sequential two-agent chats where the chat results of previous chats are cumulatively carried over to the following ones, adding more context to the chats. Any combination of agents can be used in each two-agent chat. This is shown in Figure 4.12.
3. **Group Chat:** This pattern sets up a chat between more than two agents within the same conversation, rather than being a combination of combinations of two-agent chats. The pattern is built around cycles where one agent is chosen to speak and does so, the other agents listen, and finally a new speaker is chosen to speak. This is facilitated using a special type of Agent, the Group Chat Handler. It receives the message from the speaker, it then broadcasts it to the other agents in the group chat, as they do not receive it directly from the speaker. After broadcasting the message, the Group Chat Handler selects the next speaker, starting a new cycle. This is shown in Figure 4.13. The Group Chat Handler has, at the time of writing, four supported strategies for selecting the next speaker:
   (a) Round-robin: The next speaker is selected by taking turns in a round robin arrangement [73].
   (b) Random: The next speaker is selected at random.
   (c) Manual: The Group Chat Handler selects the next speaker by asking for

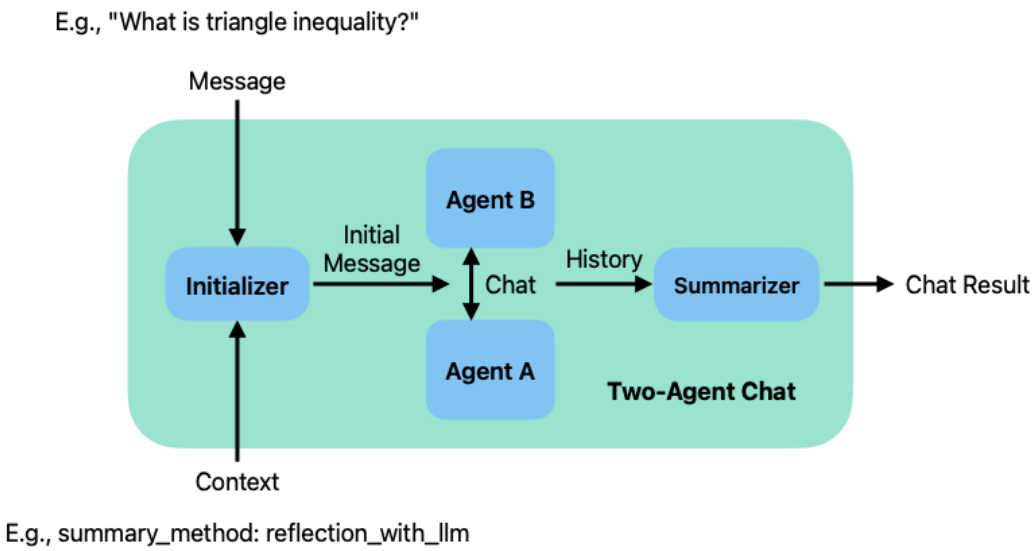

Figure 4.11: AutoGen's Two-Agent Chat conversational pattern [72].

human input.

(d) Auto: This is the default setting where the Group Chat Handler uses its LLM to select the next speaker.

4. **Nested Chat:** This is a pattern that uses nested sequential chats and routes the responses through a single agent as a conversational interface. This is facilitated through a standard pluggable component for the agents, making one agent into a nested chats handler. This is shown in Figure 4.14.

Design patterns

AutoGen follows a multi-agent architecture, but the difference to other multi-agent projects is that each agent in this architecture makes use of the RAG (Retrieval Augmented Generation) pattern in order to function. Moreover, this includes the usage of a VectorDB (ChromaDB [70]) for contextual purposes.

Comments

Whilst an attempt to contact developers or contributors on the official AutoGen Discord server was made, the only response given instructed to read the tutorial pages on their website [74]. No further responses have been given at the time of writing.

4.4.8 GPT Pilot

GPT Pilot is the core technology behind the Visual studio code extension called Pythagora [75], with the stated aim of providing the "first real AI developer companion" [76].

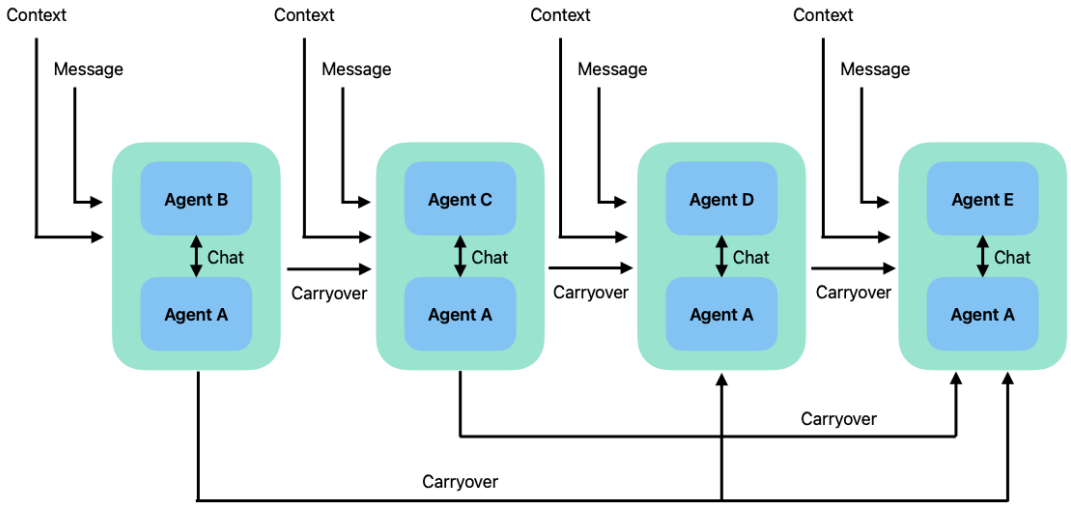

Figure 4.12: AutoGen's Sequential Chats pattern [72].

Components

GPT Pilot's main components seem to only be different kind of agents with roles mirroring those that are traditionally part of software development agencies.

Flow structure

The GPT Pilot's high level description of its flow is very similar, and intentionally so, to the flow of software development agencies, with stakeholders and developers represented by six different agents [77]. The user provides a description for an application they want developed, the product owner agent takes over, it may ask for clarifications from the user before continuing. The request, along with requirements are sent to the developer agents, including, but not limited to, the architect agent and developer agent, as shown in Figure 4.15.
GPT Pilot, being focused on development and coding, also has a way to keep track of already written code without needing to send the entire codebase to the LLM [78]. It does this via a different agent to the coding agent that tracks down relevant files, converts them into pseudocode, asks the LLM what parts of the file is relevant and finally sends the relevant code parts to the coding agent, as shown in Figure 4.16.

Design patterns

GPT Pilot follows a multi-agent framework where the agents form a sequence from one agent to another with occasional cycles in the flow. The agents are based on SDLC (Software Development Life Cycle), where each agent takes the role of an entity in the SDLC. For example, there is the Product Owner agent which talks to the client team and give specifications to the development team. The roles are assumed by the agents by simple prompt-engineering. The prompts are collected

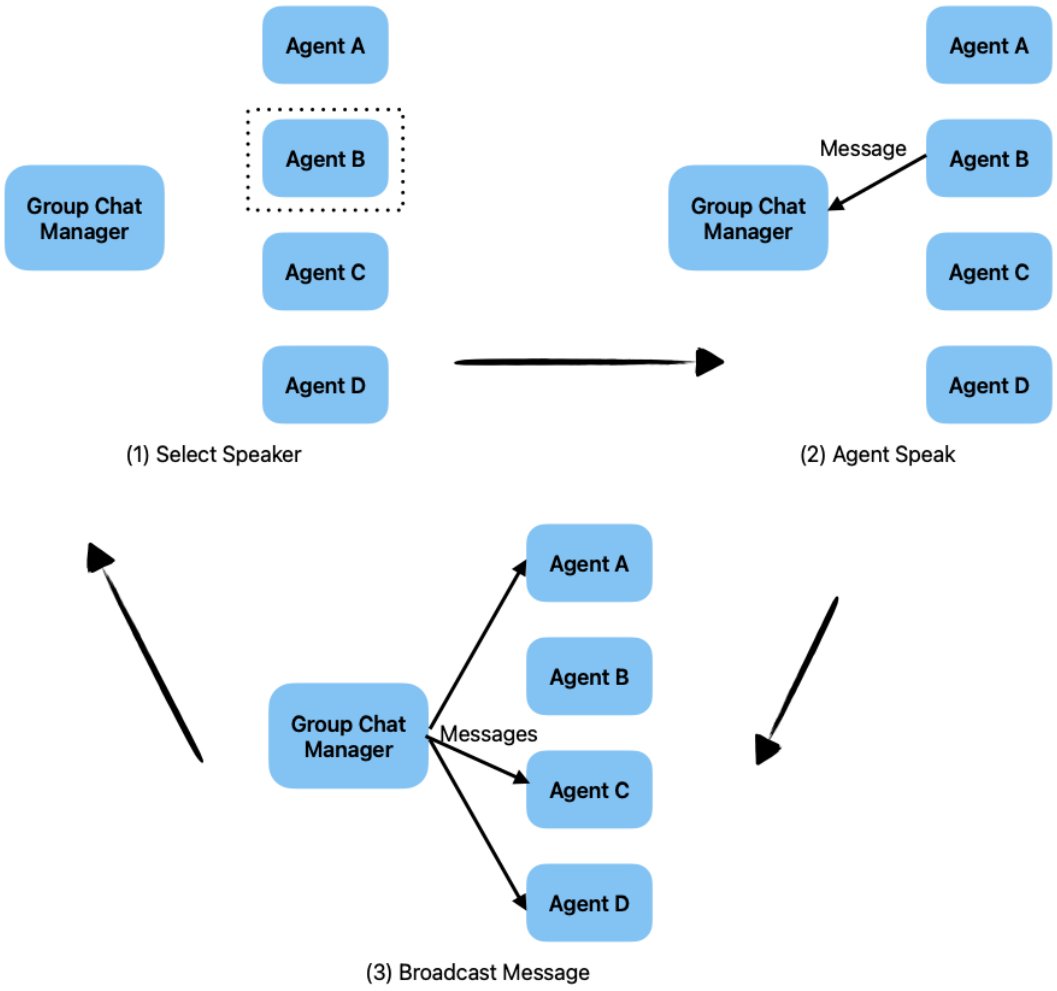

Figure 4.13: AutoGen's Group Chat Pattern [72].

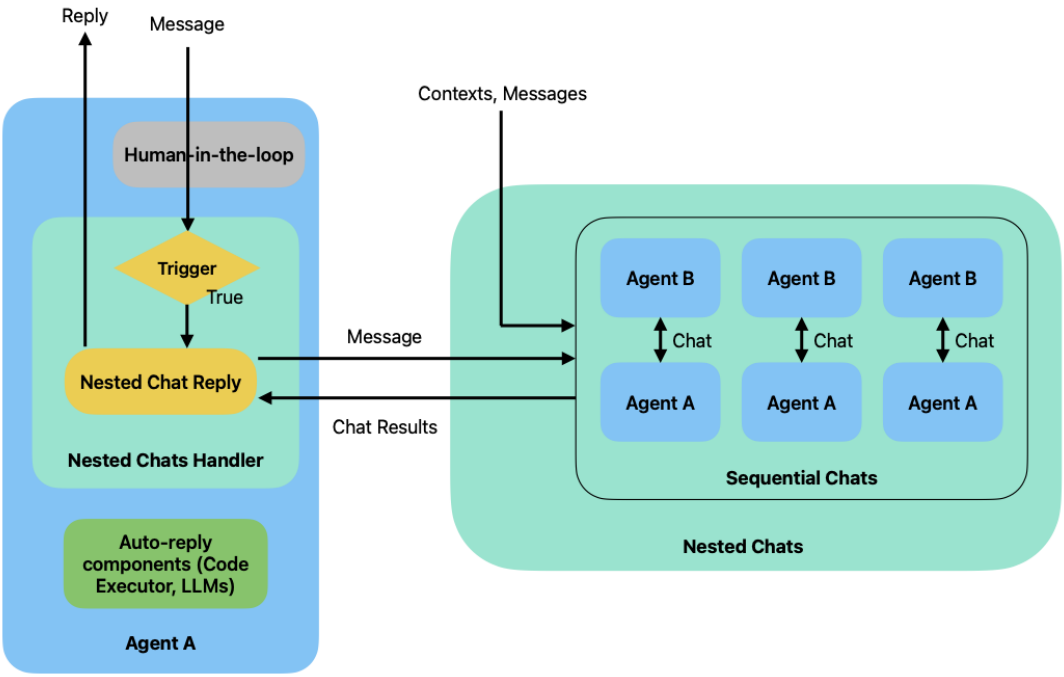

Figure 4.14: AutoGen's Nested Chats Pattern [72].

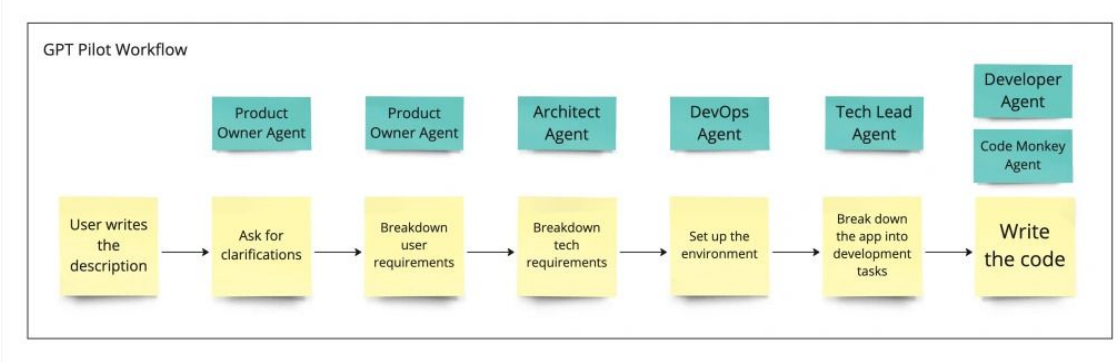

Figure 4.15: GPT Pilot's product development flow[77].

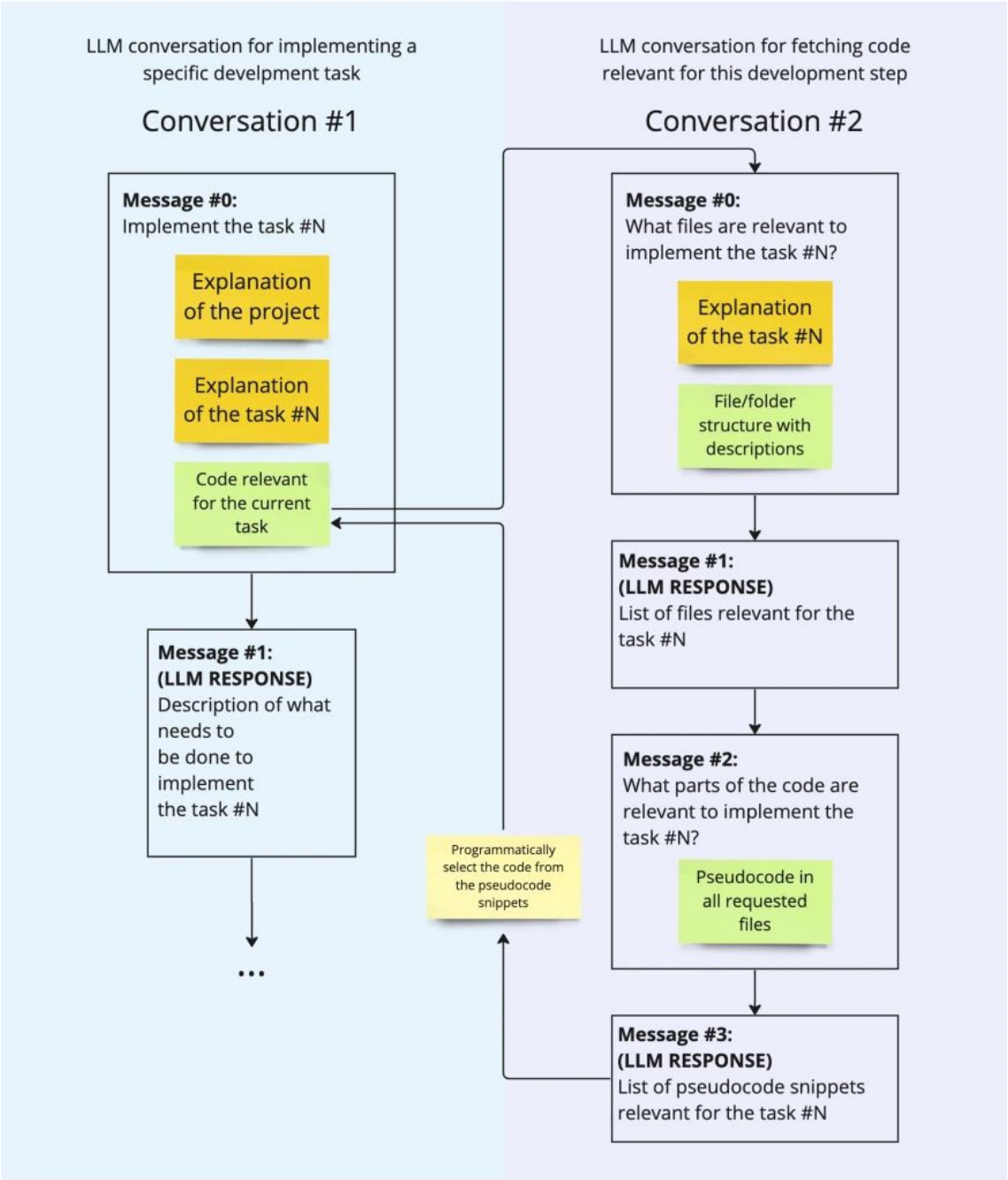

Figure 4.16: GPT Pilot's code tracking flow[78].

in the `prompts/` directory in separate `*.prompt` files. The application also uses a postgres database for storing information about the status of the application. Moreover, GPT Pilot also makes use of In-context learning in the case where it considers the state of the codebase before any coding step. The agents themselves operate in an ad-hoc architectural way where the models can be chosen from either the OpenAI or the Anthropic APIs.

Comments

Attempts at contacting developers of GPT Pilot via their official Discord server were successful. However, the response given was merely to read their development blog. The blog does contain two posts describing, in an abstract manner, how GPT Pilot functions. These blog posts are part of an, at the time of writing, incomplete three part series of blog posts describing the abstract functionality and workflow of GPT Pilot.

4.4.9 CrewAI

CrewAI is a multi-agent framework for autonomous role-playing AI-agents, with the express purpose of helping said agents cooperate to solve complex tasks [54].
CrewAI is built around several core concepts.

- Agents
- Tasks
- Tools
- Processes
- Crews
- Collaboration
- Memory

Components

The following core concepts of CrewAI are component related, or component adjacent:

- **Agents:** CrewAI's agents are built to be able to, out of the box, perform tasks, make decisions, and communicate with other agents [79]. The only things that they require to function are their role, goal, and backstory. They can, however, be customised further by providing a list of tools it can use, select what LLM it will use, customising the maximum amount of iterations the agent is allowed to perform, amongst many optional parameters.
- **Tasks:** Tasks are created by specifying what agent is going to perform it, the description of the task, and the expected output [80]. Tasks can also be given specific tools to be used to complete it. They can also be customised to be asynchronous, meaning that other agents won't wait for this task to be completed before continuing with their next text.
- **Tools:** CrewAI has several built in tools, most of them being search tools for different websites and file formats [59]. Custom and LangChain tools are also supported.

- **Processes:** Processes in the context of CrewAI is referring to built in task execution strategies [81]. There are, at the time of writing, two processes built-in:
  - *Sequential:* Tasks are completed sequentially.
  - *Hierarchical:* A manager LLM model is specified to delegate and organise the task execution order based on a structured chain of command.
- **Crews:** A crew in CrewAI is a collection of agents working together to complete a set of tasks specified alongside the agents when creating it [82]. Crews use the different aforementioned processes to structure their task execution strategies.
- **Collaboration:** Collaboration between agents in CrewAI is centered around three fundamental things [83]:
  - Agents share information and findings to make sure all agents are well informed.
  - Agents are able to ask for help from other agents when dealing with things outside of their area of expertise.
  - Agents share resources efficiently which optimizes the execution of tasks.
- **Memory:** Similarly to DroidAgent, CrewAI uses different memory types [84]. Unlike DroidAgent, however, it uses four types of memory rather than three:
  - *Short term memory:* Temporary storage of recent interactions.
  - *Long term memory:* Preserves insights and learned information for refining an agent's knowledge.
  - *Entity memory:* Organises entity information, such as people and concepts, for deeper understanding and relationship mapping.
  - *Contextual memory:* For aiding in the coherency of agent responses by maintaining the context of interaction.

  All of the memory types uses embeddings, with OpenAI embeddings being the default.

Flow structure

CrewAI is structured in the following way. Initially, agents are customized and created to solve or focus on a particular task. The agents can also be equipped with tools to enhance their abilities further and provide accurate responses. Each task is assigned an agent and multiple agents can be ordered sequentially or a manager agent can be assigned to overlook the work done by other agents. The manager agent also reviews the overall work and provides the best output to the user.

Design patterns

CrewAI makes use of a multi-agent architecture where each agent executor, parser and tools_handler makes use of agents from langchain. Each agent in the architecture makes use of RAG architecture to function. Moreover, the agents interact with each other based on the required task. Moreover, this project also boasted four different types of memories long, short, contextual and entity memory to handle different nuances in the task.

Comments

CrewAI's customizable role-based agents sparked discussion about meta-creation. In this context, meta-creation refers to the concept of an agent being provided a tool, which can create an agent. So the system can generate the required agents automatically, collaborate with it and provide relevant output to the users. This is not mentioned in the documentation, but should rather be treated as food for thought.

4.5 Cognitive architectures

This section displays the results achieved by analysis on cognitive architectures concerning LLMs. In this section, cognitive architectures are described in a general sense, which will be related to the current research., in further sections.

4.5.1 Traces in LLM based applications

This subsection discusses the various components or traces of cognitive architectures that were observed in the set of LLM based applications. It is to be noticed that the selected projects are purely software based and do not rely on perception as a cognitive entity. To overcome this, perception here is simply assumed to be the input data provided in the form of natural language. Thus it is safe to assume that perception is an available feature in all the selected projects. It is also crucial to notice that not a lot of projects emphasize the explicit usage of cognitive architectures but rather appear to use cognitive architectures in one form or another.

MetaGPT

In MetaGPT, there were a few observations made which could establish potential connections to cognitive architectures. MetaGPT makes use of both long and short term memories. In MetaGPT, each agent created has a memory component that stores every bit of message that it observes. This can be understood as the agent's short-term memory. When recorded memories are needed, such as serving as context for a LLM call, the agent can retrieve the most recent memories or all memories. However, there are multiple branches of memories as well as a wide range of memory generation and retrieval techniques. Long-term memory in MetaGPT is used to remember the short-term memory (STM) news and integrate the STM news with long-term memory (LTM) news.
Since MetaGPT is a multi-agent collaborative framework, it is observed that the agents interact with each other to form a distributed network. This can be observed in cognitive architectures where different cognitive agents collaborate towards a common task / goal. The primary goal as observed in MetaGPT was to encode human SOPs into LLM agents. This enables the system to learn new experiences and adapt to varying SOPs. MetaGPT can solve problems which aligns with the reasoning based on learned knowledge.
Although the architecture used in the project cannot be boiled down to one, there are traces of a few architectures. In one aspect, it can even be observed that the

subsumption architecture which breaks down complex problems into simple robotic activities is a behaviour demonstrated by the agents, although there isn't hardware robots involved. Moreover, MetaGPT has not officially stated the usage of cognitive architectures in the project, as cognitive architectures require demonstration of a full-range of human cognition. But in general it was observed that individual components of cognitive architectures can be identified, however those components do not function in collaboration to contribute a unique architecture.

DroidAgent

DroidAgent makes use of long, spatial and short term memories in it's architecture. The usage of short and long term memory suggest traces to the Soar cognitive architecture for this use-case. In this case, the working memory, or short term memory stores the current and previous GUI state, and the current task, thus maintaining the agent's situational awareness. Although the research doesn't make use of an out-of-the-box cognitive architecture, there are aspects of the research which can be seen as related to cognitive architectures. This is significantly seen by the goal or intention oriented behaviour of the individual agents, which are similar to that of Soar and ACT-R. The learning aspect of the research can be considered as a subset of traditional architectures which use long and short term memories to adapt and learn to a situation. Moreover, the concept of "self-critique" in DroidAgent adds to the meta-reasoning concept of cognition [85].

Continue

Upon analysis of Continue, traces of cognitive architectures were found. Code-refactoring is a major feature of Continue, which can be seen as reasoning in the cognitive architecture entity list. According to TechTarget, refactoring is a process where a piece of code is restructured without changing the intended behaviour of the code. This is usually done as a part of code optimization [86]. This phase involves problem-solving and decision-making which are crucial for reasoning. The project system is supposed to reason about the current state of the code and the desired state and how best it can get there. Thus refactoring code can be seen as reasoning. Moving on, Continue also has a short term memory in action considering the need of a context to perform the "In-context learning" process. However, these are observations and have not been explicitly mentioned anywhere in the documentation.

InvokeAI

InvokeAI makes use of calls to the HuggingFace API to perform the required tasks. There is no other common entity to cognitive architectures as this project only relies on API calls.

BabyAGI

BabyAGI has a multi-agent architecture primarily with calls to the OpenAI API to support its individual agents. According to an online article by Dan the AI Man, it is observed that the roadmap for the BabyAGI project did consist of plans to

incorporate cognitive architectures so the system could learn, adapt and reason [87]. However, upon further inspection of code, it is observed that there is no presence of reasoning and problem solving entities. There is no explicit trace of cognitive architecture observed nor something mentioned in the documentation.

Promptify

As mentioned in subsection 4.4.3, promptify is primarily a static tool and is an outlier among the selected projects. There is no observed cognitive architecture in the project.

AutoGen

AutoGen is a framework developed by Microsoft, which makes use of customizable and conversable agents. According to Wu, Bansal, Zhang, *et al.*, the agents in AutoGen can not only converse with humans but also interact with each other [17]. Moreover, the agents can also reason with each other to support their reasoning. This introduces a new concept called meta-reasoning (a subset of meta-cognition) where one can reason about their own reasoning processes [85]. Furthermore, the collaborative effort of the agents display distributed cognition towards a common task or goal. Finally, it is also observed that AutoGen makes use of short term memory in terms of cache, and a long-term memory as a VectorDB to retrieve context for different situations.

GPTPilot

GPTPilot makes use of a database (PostGres) as a long-term memory for history while user entries are being stored in the short-term memory. Moreover, similar to Continue, GPTPilot can also reason as it can refactor code while also discuss with the user about issues and new pull requests which emphasize comprehension as a cognitive entity. Moreover, the communication between the agents of the multi-agent architecture gives rise to collaboration in problem solving.

CrewAI

CrewAI uses both long and short term memory, on top of entity memory and contextual memory. The long-term memory aids the agents to learn from previous experiences while the short term memory aids it to recall recent conversations. Moreover, information is also shared between the agents which helps the agents to collaborate to achieve a common goal. An additional functionality called delegation, allows the agents to intelligently assign tasks to different agents which optimizes the crew's overall efficiency. This can be seen in comparison to the blackboard architecture [28].

4.6 Rejected Projects

This section displays the different rejected projects and the motivation behind their rejection.

Table 4.5: Rejected Projects and Motivation for Rejection

Rejected project	Motivation
Autolabel	Autolabel's Discord server was completely inactive; the only activity seemed to be people joining the server.
AutoGPT	Similar to Autolabel, the developers did not respond to any messages posted in any of the Discord channels.
GPT4All	The GPT4All Discord server was very active. Unfortunately, nobody, including developers or contributors, responded to the interview queries directed at them.
RasaGPT	RasaGPT was considered but rejected because the documentation required running the application, which was unsuccessful. Additionally, there was no reliable way to contact the project's developer.
GPT-Engineer	GPT-Engineer showed initial promise. However, an interview with a developer representative revealed that the developers lacked structural documentation and were unable to explain it to the interviewer.
PR-Agent	The documentation of PR-Agent did not include information needed for this research. When asked, the developers did not provide very useful information and implied that they would not provide such information because of secrecy concerns.

5
Discussion

5.1 Result Summary

This section includes a summary of the results from the previous section. In summary, the results show that there are observable structures that can be extracted into patterns in LLM based open source software projects. Using the literature review, we got a clear scope of the existing research in the domain. This in turn helped us to adjust and pivot the scope of our research. The review of existing literature provided valuable insights into the employment of different design patterns for different use cases and the overall adaptability of the multi-agent architecture for different scenarios. The results also show that there are identifiable traces of, or commonalities with, cognitive architectures within the observed architectures in the different projects. Some projects were also observed to have structures that were similar to, and could be improved with, established software patterns. In the end, a library of 12 LLM design patterns that have been identified, described, and named in the results has been assembled. Having said this, we attempt to answer the research questions with the results we achieved.
Table 5.1 displays the various results obtained from this research and their connection to the initially posed research questions.

5.2 Contributions

This section explains the different contributions this study has made in the domain of software architectures for LLM based applications. It provides a comprehensive analysis of current architectural patterns utilized in integrating Large Language Models (LLMs) into software applications. This is done by both an extensive literature review of the domain and by examining a variety of open-source projects. Using this information, we were able to identify design patterns and their applicability, thus offering a valuable resource for developers seeking to implement LLM based use cases.
Unlike existing studies that focus on specific aspects or applications of LLMs, our research provides a thorough analysis of the architectural patterns used in integrating LLMs into software applications. This includes not only identifying 12 common design patterns but also categorizing them within the well-established framework of the Gang of Four's design patterns, tailored specifically for LLM-based applications [11]. This library serves as a standardized reference for addressing common design challenges in LLM integration, promoting best practices, and enhancing the

Table 5.1: Research Questions and Corresponding Results

Research Question	Result
RQ1: What literature is available regarding architectures for LLM integration in an application?	Comprehensive literature review identified different ways to integrate LLMs to a use-case. In-context learning, RAG and variations of Multi-agent architectures and many more intricate design patterns mentioned in the results, contribute to different methods LLMs can be incorporated with an application.
RQ2: Are there identifiable design patterns used in open source software development with LLM integrations?	Analysis of open-source projects revealed patterns such as Multi-Agent (Role-based, Hierarchical, Sequential, Cyclic, Parallel), Retrieval-Augmented Generation (RAG), and In-context learning. We generated a library of design patterns to show that there are indeed identifiable patterns in open-source software development.
RQ3: How can integrating cognitive architectures be helpful for LLM dependent applications?	Evaluating projects like MetaGPT, DroidAgent, and Continue demonstrated that cognitive architectures can enhance efficiency and functionality by simulating human cognition processes.

robustness and scalability of these applications.
Furthermore, the study explores the role of cognitive architectures within LLM-based systems, identifying key components and their interactions. This examination not only highlights the potential for cognitive architectures to improve the efficiency and functionality of LLM applications but also suggests pathways for future research and development in this emerging domain. This exploration is relatively novel since most of the research in cognitive architectures has been done concerning robotic process automation. So this research adds a new dimension to the understanding of LLM integration, bridging a gap in current literature that often overlooks the potential of cognitive architectures in this context.
Lastly, the empirical insights gathered from developer interviews enrich the theoretical findings, ensuring that the proposed patterns and architectures are grounded in practical experience. This approach bridges the gap between academic research and real-world application, contributing to the broader discourse on software architecture in the age of advanced AI technologies.

5.3 Prompt engineering

An observation was made on the grounds of how prompt engineering was carried out for the ad-hoc methods for different project agents. Some of the above-mentioned

projects, leveraged prompt generation to control the output of the LLM to a specific format. This helped to create a pipeline between an LLM instance to another or a different piece of code. Promptify is an analysed project, which made use of Jinja files to generate prompts which are then fed into an LLM instance to get highly controlled outputs. This sparks a discussion about the need for prompt engineering in itself. The purpose of LLMs in general was to comprehend and process natural language which should require less to no engineering required. The answer lies in the difference between using an LLM versus exploiting its maximum potential.

5.4 Analysis of Multi-agent architectures

From the analysis of the projects, it is observed that larger-scale projects tend to gravitate towards Multi-agent architecture for their needs. Another observation was of the nature of the agents themselves. It was interesting to notice that while a lot of agents were built using ad-hoc methods (as in each agent making API calls to a LLM provider), AutoGen made use of RAG architecture for each of its agents. The prevalence of unstructured or undocumented codebases seems to be high in this area of study, as demonstrated by projects such as MetaGPT (in Figure4.1), Continue (in A.2.1.1), and the ones found in A.2.2. This can likely be attributed to the relatively new area of development, experimental coding taking precedence over structure.
The usage of multi-agent architectures for scalable projects is justified by the nature of the architecture itself, in a way that the agents can interact with each other in multiple unique ways to produce different results, that can be tailored to unique use cases. As mentioned by Li, Zhang, Yu, *et al.*, in their paper titled "More Agents Is All You Need" multiple agents and different ways of interaction between said agents help in scaling LLM based applications [30]. An agent, in this context can be a simple call to the LLM or a sophisticated design pattern in itself like RAG, with dedicated memory modules. Another observation was made about the similarity between the multi-agent architecture and its potential to be a blackboard architecture for LLM based applications [28]. A blackboard architecture is similar to a multi-agent where each agent acts autonomously to solve a common goal or problem based on their roles. This aspect of different agents to work towards a common goal also adds to the multi-agent framework being relevant to cognitive architectures. From the analysed projects, it was observed that most of the projects which made use of multi-agent architectures contained traces of cognitive architectures or had the potential to be developed into a cognitive architecture.

5.5 Cognitive architectures

It was also observed that most of the analysed applications also leveraged long and short term memories in their architectures. The presence of long-term memories indicated that the agent was capable of learning from past experiences. The study of cognitive architectures helped explore different concepts like meta-reasoning where two LLM agents reason with each other and validates the other's reasoning. However, it was observed that only BabyAGI and MetaGPT, acknowledged the use

of cognitive architectures explicitly, while the other projects make use of cognitive architectures unintentionally. This proved to be a crucial part of the study as it explored different ways to achieve AGI (Artificial General Intelligence). With this discussion, we can finally draw up a conclusion to our research.

5.5.1 Meta-reasoning

An interesting concept introduced is meta-reasoning within cognitive architectures. This involves individual agents assessing and refining each other's reasoning processes, ensuring a robust and dynamic decision-making framework. This insight not only enriches the architectural design of LLM applications but also propels further investigations into the synergistic potential of multi-agent systems and cognitive architectures, aiming towards advancements in Artificial General Intelligence (AGI).

5.6 Threats to validity

The research employed a formal qualitative methodology, which mitigates potential threats to its validity. Interviews constituted the only dynamic component, where responses varied depending on the interviewees. However, the bulk of the analysis was conducted manually, with interviews serving as a supplementary tool to clarify complex project structures. It is worth noting that there might be an issue of external validity since the nine analyzed projects may not fully represent the entire spectrum of LLM-based applications. Additionally, the rapid evolution of LLM technologies poses a risk to the temporal validity of the findings. Despite these considerations, the research maintained objectivity, ensuring no internal validity threats due to bias.

6
Conclusion

6.1 Integration of LLMs into Software Architectures

The research undertaken provides significant insights into the integration of Large Language Models (LLMs) into various software architectures. These architectures, with their decentralized and collaborative approach, offer a robust framework for handling complex and varying requirements. In contrast, smaller projects have been found to benefit from more ad-hoc architectures, leveraging techniques such as prompt engineering to manage and optimize LLM outputs effectively, as seen in the Jinja template usage in the Promptify project. Moreover, beyond just classifying the projects based on architectures, there is also a classification of the architectures themselves based on the level of depth they impact.

6.2 Multi-Agent Systems and Blackboard Architectures

A notable observation from the study is the similarity between multi-agent systems and blackboard architectures. Both frameworks involve decentralized agents contributing specialized knowledge towards a common objective, enhancing the overall system functionality. The study also highlights the role of cognitive architectures within these environments. Projects utilizing multi-agent approaches often integrate cognitive architecture aspects, such as simulating long-term and short-term memory and reasoning capabilities, thus enriching the system's operational depth.

6.3 Overall Summary

In summary, the research questions have been thoroughly addressed. The extensive literature review conducted has provided answers to RQ1. For RQ2, it has been found that design patterns in LLM-based open-source applications have distinct characteristics that are not fully described by traditional design patterns. Although the overall project may employ conventional patterns like the modular pattern, these do not satisfy the LLM-specific aspects, which are the central focus of this study. There are observable similarities between traditional patterns and those designed for LLM applications. For instance, the Observer pattern, which involves a "sub-

ject" notifying its "observers" of state changes, can be compared to the multi-agent architecture in LLM applications, where agents communicate towards a common goal. However, this comparison is not exact. Regarding RQ3, the research suggests that integrating cognitive architectures could potentially lead to the development of AGI.

6.4 Future Work

Having concluded the research, this section discusses the potential future work in the area. Advancements are being made concerning design patterns for LLM based applications. According to Borek, concepts like GuardRails, Responsible AI, Advanced RAG are being introduced which would lead to newer design patterns to include in the library of patterns [88]. Moreover, fine-tuning as a pattern was also omitted since it did not contribute to be a design pattern but rather an optimisation of the LLM itself. Henceforth, future work would reasonably include a wider study with more projects and more time to analyse them, preferably with experts within the field of software design patterns. More work could also be made to analyse the efficiency of patterns for LLM applications, to see how well they scale and perform in different contexts. Another aspect of this could also be to compare general software architectures and patterns with ones specialised for LLMs to see if the specialised ones are more efficient.

Bibliography

[1] B. Marr, *A short history of chatgpt: How we got to where we are today*, 2023. [Online]. Available: https://www.forbes.com/sites/bernardmarr/2023/05/19/a-short-history-of-chatgpt-how-we-got-to-where-we-are-today/?sh=2cedb0dd674f.

[2] Welocalize, *AI & LLMS: TRANSFORMING BUSINESS INNOVATION*, 2023. [Online]. Available: https://www.welocalize.com/ai-llms-transforming-business-innovation/ (visited on 11/06/2023).

[3] B. Meskó, "Prompt engineering as an important emerging skill for medical professionals: Tutorial," *J Med Internet Res*, vol. 25, e50638, Oct. 2023, ISSN: 1438-8871. DOI: 10.2196/50638. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/37792434.

[4] S. Hong, M. Zhuge, J. Chen, *et al.*, *Metagpt: Meta programming for a multi-agent collaborative framework*, 2023. arXiv: 2308.00352 [cs.AI].

[5] F. Brooks Jr, "No silver bullet essence and accidents of software engineering," *IEEE Computer*, vol. 20, pp. 10–19, Apr. 1987. DOI: 10.1109/MC.1987.1663532.

[6] A. Butani, *5 essential patterns of software architecture*, 2020. [Online]. Available: https://www.redhat.com/architect/5-essential-patterns-software-architecture (visited on 11/06/2023).

[7] A. LaFrance, *The story behind twitter's fail whale*, https://www.theatlantic.com/technology/archive/2015/01/the-story-behind-twitters-fail-whale/384313/, 2015. (visited on 11/13/2023).

[8] Turing, *How llms are changing the face of business analytics*, 2023. [Online]. Available: https://www.turing.com/resources/how-llms-are-changing-the-face-of-business-analytics (visited on 11/06/2023).

[9] Z. Yan, *Patterns for building llm-based systems & products*, 2023. [Online]. Available: https://eugeneyan.com/writing/llm-patterns/ (visited on 02/08/2024).

[10] J. Yoon, R. Feldt, and S. Yoo, "Autonomous large language model agents enabling intent-driven mobile gui testing," *arXiv preprint arXiv:2311.08649*, 2023.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[12] I. A. Blank, "What are large language models supposed to model?" *Trends in Cognitive Sciences*, vol. 27, no. 11, pp. 987–989, 2023.

[13] IBM, *What are ai hallucinations?* 2023. [Online]. Available: https://www.ibm.com/topics/ai-hallucinations (visited on 05/02/2024).

[14] Z. Ji, N. Lee, R. Frieske, *et al.*, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, Mar. 2023, ISSN: 1557-7341. DOI: 10.1145/3571730. [Online]. Available: http://dx.doi.org/10.1145/3571730.

[15] IBM, *What is overfitting?* 2021. [Online]. Available: https://www.ibm.com/topics/overfitting (visited on 05/03/2024).

[16] W. D. Heaven, *Google deepmind wants to define what counts as artificial general intelligence*, 2023. [Online]. Available: https://www.technologyreview.com/2023/11/16/1083498/google-deepmind-what-is-artificial-general-intelligence-agi/ (visited on 05/07/2024).

[17] Q. Wu, G. Bansal, J. Zhang, *et al.*, "Autogen: Enabling next-gen llm applications via multi-agent conversation framework," 2023. arXiv: 2308.08155 [cs.AI].

[18] N. Choi, *The architecture of today's llm applications*, 2023. [Online]. Available: https://github.blog/2023-10-30-the-architecture-of-todays-llm-applications/ (visited on 11/15/2023).

[19] C. Stryker, M. Scapicchio, and IBM, *What is generative-ai?* 2024. [Online]. Available: https://www.ibm.com/topics/generative-ai (visited on 05/07/2024).

[20] F. García-Peñalvo and A. Vázquez-Ingelmo, "What do we mean by genai? a systematic mapping of the evolution, trends, and techniques involved in generative ai," *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 8, pp. 7–16, Aug. 2023. DOI: 10.9781/ijimai.2023.07.006.

[21] D. Foster, *Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play*. O'Reilly Media, 2019, ISBN: 9781492041917.

[22] J. Barnard and IBM, *What is embedding?* 2023. [Online]. Available: https://www.ibm.com/topics/embedding (visited on 05/07/2024).

[23] C. Song and A. Raghunathan, *Information leakage in embedding models*, 2020. arXiv: 2004.00053 [cs.LG].

[24] F. Franco, *What is "in context learning" (icl)? how does it work?* 2024. [Online]. Available: https://medium.com/@francescofranco_39234/in-context-learning-icl-a775cd8b7261 (visited on 05/09/2024).

[25] D. Shah, *What is in-context learning, and how does it work: The beginner's guide*, 2023. [Online]. Available: https://www.lakera.ai/blog/what-is-in-context-learning (visited on 05/09/2024).

[26] K. Martineau, *What is retrieval-augmented generation?* 2023. [Online]. Available: https://research.ibm.com/blog/retrieval-augmented-generation-RAG (visited on 05/08/2024).

[27] IEEE Computer Society, *What is software architecture in software engineering?* 2023. [Online]. Available: https://www.computer.org/resources/software-architecture (visited on 05/10/2024).

[28] B. Hayes-Roth, "The blackboard architecture: A general framework for problem solving?" Heuristic Programming Project, Stanford University, Tech. Rep. HPP-83-30, 1983.

[29] A. Bondi, "Characteristics of scalability and their impact on performance," Sep. 2000, pp. 195–203. DOI: 10.1145/350391.350432.

[30] J. Li, Q. Zhang, Y. Yu, Q. Fu, and D. Ye, "More agents is all you need," *arXiv*, vol. abs/2402.05120, 2023.

[31] S. Ryu, *Emerging architectures for llm applications*, 2023. [Online]. Available: https://a16z.com/emerging-architectures-for-llm-applications/ (visited on 07/01/2024).

[32] L. Heiland, M. Hauser, and J. Bogner, "Design patterns for ai-based systems: A multivocal literature review and pattern repository," *arXiv preprint arXiv:2303.13173*, 2023.

[33] T. Tung, *7 architecture considerations for generative ai*, 2023. [Online]. Available: https://www.accenture.com/us-en/blogs/cloud-computing/7-generative-ai-architecture-considerations (visited on 07/01/2024).

[34] E. Yan, *Patterns for building llm-based systems & products*, 2023. [Online]. Available: https://eugeneyan.com/writing/llm-patterns/ (visited on 07/01/2024).

[35] S. Van Vaerenbergh, *Awesome generative ai*, 2023. [Online]. Available: https://github.com/steven2358/awesome-generative-ai (visited on 07/01/2024).

[36] G. Marcus, E. Leivda, and E. Murphy, "A sentence is worth a thousand pictures: Can large language models understand human language?" *arXiv preprint arXiv:2308.00109*, 2023.

[37] H. Chase, *Openai's bet on a cognitive architecture*, 2024. [Online]. Available: https://blog.langchain.dev/openais-bet-on-a-cognitive-architecture/ (visited on 07/01/2024).

[38] Hugging Face, *The ai community building the future*, 2024. [Online]. Available: https://huggingface.co (visited on 05/12/2024).

[39] Anthropic, *Getting started - claude - anthropic*, 2024. [Online]. Available: https://www.anthropic.com/api (visited on 07/01/2024).

[40] LangChain, *Langchain*, 2023. [Online]. Available: https://www.langchain.com/ (visited on 07/01/2024).

[41] Hugging Face, *Retrieval-augmented generation for llms*, https://huggingface.co/blog/retrieval-augmented-generation, 2023. (visited on 05/12/2024).

[42] Oracle, *Enhancing ai with in-context learning*, 2023. [Online]. Available: https://www.oracle.com/a/ocom/docs/cloud/in-context-learning-ai.pdf (visited on 07/01/2024).

[43] Azure, *Using apis for llm integration*, 2023. [Online]. Available: https://github.com/Azure/APIs-for-LLM (visited on 07/01/2024).

[44] Azure, *Implementing multi-agent systems for scalable ai solutions*, 2023. [Online]. Available: https://github.com/Azure/Multi-Agent-Architectures (visited on 07/01/2024).

[45] C. Greyling, *Retrieval-augmented generation (rag) vs llm fine-tuning*, https://cobusgreyling.medium.com/retrieval-augmented-generation-rag-vs-llm-fine-tuning-3f311211919a, 2023. (visited on 07/01/2024).

[46] G. Wrba, *Exploring llms using openai api*, 2024. [Online]. Available: https://www.linkedin.com/pulse/exploring-llms-using-openai-api-guillermo-wrba-isidc (visited on 07/01/2024).

[47] A. Pal, *Promptify: Structured output from llms*, 2022. [Online]. Available: https://github.com/promptslab/Promptify (visited on 07/01/2024).

[48] J. R. Anderson, M. Matessa, and C. Lebiere, "ACT-R: A theory of higher level cognition and its relation to visual attention," *Human–Computer Interaction*, vol. 12, no. 4, pp. 439–462, 1997. DOI: 10.1207/s15327051hci1204_5. [Online]. Available: https://doi.org/10.1207/s15327051hci1204_5.

[49] P. Lewis, E. Perez, A. Piktus, *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *arXiv preprint arXiv:2005.11401*, 2020.

[50] Y. Hoshi, D. Miyashita, Y. Ng, *et al.*, *Retrieval-augmented generation for large language models: A survey*, https://arxiv.org/html/2312.10997, 2023. (visited on 05/27/2024).

[51] T. B. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[52] OpenAI, *Openai api*, https://beta.openai.com/docs/, 2020. (visited on 05/27/2024).

[53] E. H. Durfee, "Multiagent systems: A modern approach to distributed artificial intelligence," *Artificial Intelligence*, vol. 110, no. 1-2, pp. 145–147, 1999.

[54] crewAI, *crewAI*, 2024. [Online]. Available: https://github.com/joaomdmoura/crewai/ (visited on 05/19/2024).

[55] L. Gasser, "Distributed artificial intelligence," in *Distributed artificial intelligence*, Elsevier, 1987, pp. 37–40.

[56] V. De Florio and C. Blondia, "The master-slave paradigm in distributed computer systems," *Journal of Systems Architecture*, vol. 41, no. 4, pp. 293–310, 1995.

[57] DeepLearning.AI, *Agentic design patterns part 3: Tool use*, 2023. [Online]. Available: https://www.deeplearning.ai/the-batch/agentic-design-patterns-part-3-tool-use/ (visited on 07/01/2024).

[58] M. Fowler, *Engineering practices for llm application development*, 2023. [Online]. Available: https://martinfowler.com/articles/engineering-practices-llm.html (visited on 07/01/2024).

[59] crewAI, *Tools*, 2024. [Online]. Available: https://docs.crewai.com/core-concepts/Tools/ (visited on 05/21/2024).

[60] J. E. Laird, *Introduction to soar*, 2022. arXiv: 2205.03854 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2205.03854.

[61] P. Langley, J. E. Laird, and S. Rogers, "Cognitive architectures: Research issues and challenges," *Cognitive Systems Research*, vol. 10, no. 2, pp. 141–160, 2009, ISSN: 1389-0417. DOI: https://doi.org/10.1016/j.cogsys.2006.07.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389041708000557.

[62] H.-Q. Chong, A.-H. Tan, and G.-W. Ng, "Integrated cognitive architectures: A survey," *Artificial Intelligence Review*, vol. 28, pp. 103–130, Jan. 2007. DOI: 10.1007/s10462-009-9094-9.

[63] MetaGPT, *Concepts*, 2023. [Online]. Available: https://docs.deepwisdom.ai/main/en/guide/tutorials/concepts.html (visited on 03/25/2024).

[64] B. Rekiek and A. Delchambre, *Assembly Line Design: The Balancing of Mixed-Model Hybrid Assembly Lines with Genetic Algorithms* (Springer Series in Advanced Manufacturing). Springer London, 2006. DOI: 10.1007/b138846. [Online]. Available: https://link.springer.com/book/10.1007/b138846.

[65] MetaGPT, *Multiagent 101*, 2024. [Online]. Available: https://docs.deepwisdom.ai/main/en/guide/tutorials/multi_agent_101.html (visited on 05/12/2024).
[66] Jinja, *Jinja*, 2007. [Online]. Available: https://palletsprojects.com/p/jinja/ (visited on 03/25/2024).
[67] promptslab, *Medical$_n$er.py*, 2023. [Online]. Available: https://github.com/promptslab/Promptify/blob/main/examples/medical_ner.py (visited on 03/25/2024).
[68] Y. Nakajima, *Babyagi*, 2023. [Online]. Available: https://github.com/yoheinakajima/babyagi (visited on 04/15/2024).
[69] Weaviate, *The ai-native vector database*, 2024. [Online]. Available: https://weaviate.io/platform (visited on 05/27/2024).
[70] Chroma, *Chroma: The ai-native open-source embedding database*, 2024. [Online]. Available: https://github.com/chroma-core/chroma (visited on 05/12/2024).
[71] Microsoft, *Introduction to autogen*, 2024. [Online]. Available: https://microsoft.github.io/autogen/docs/tutorial/introduction (visited on 04/22/2024).
[72] Microsoft, *Conversation patterns*, 2024. [Online]. Available: https://microsoft.github.io/autogen/docs/tutorial/conversation-patterns (visited on 04/22/2024).
[73] K. T. Hanna, *What is round robin?* 2022. [Online]. Available: https://www.techtarget.com/whatis/definition/round-robin (visited on 05/19/2024).
[74] Microsoft, *Tutorial*, 2024. [Online]. Available: https://microsoft.github.io/autogen/docs/tutorial (visited on 04/22/2024).
[75] Pythagora, *Pythagora (GPT pilot) beta*, 2024. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=PythagoraTechnologies.gpt-pilot-vs-code& (visited on 05/11/2024).
[76] Pythagora, *GPT pilot*, 2024. [Online]. Available: https://github.com/Pythagora-io/gpt-pilot?tab=readme-ov-file (visited on 05/11/2024).
[77] Pythagora, *GPT pilot – a dev tool that speeds up app development by 20x*, 2023. [Online]. Available: https://blog.pythagora.ai/2023/08/23/430/ (visited on 05/11/2024).
[78] Pythagora, *GPT Pilot – a dev tool that writes 95% of coding tasks [Part 2/3 – Coding Workflow]*, 2023. [Online]. Available: https://blog.pythagora.ai/2023/09/04/gpt-pilot-coding-workflow-part-2-3/ (visited on 05/11/2024).
[79] crewAI, *Agents*, 2024. [Online]. Available: https://docs.crewai.com/core-concepts/Agents/ (visited on 05/21/2024).
[80] crewAI, *Tasks*, 2024. [Online]. Available: https://docs.crewai.com/core-concepts/Tasks/ (visited on 05/21/2024).
[81] crewAI, *Processes*, 2024. [Online]. Available: https://docs.crewai.com/core-concepts/Processes/ (visited on 05/21/2024).
[82] crewAI, *Crews*, 2024. [Online]. Available: https://docs.crewai.com/core-concepts/Crews/ (visited on 05/21/2024).
[83] crewAI, *Collaboration*, 2024. [Online]. Available: https://docs.crewai.com/core-concepts/Collaboration/ (visited on 05/21/2024).

[84] crewAI, *Memory*, 2024. [Online]. Available: https://docs.crewai.com/core-concepts/Memory/ (visited on 05/21/2024).

[85] A. L. Brown, "Knowing when, where and how to remember: A problem of metacognition," in *Advances in Instructional Psychology*, R. Glaser, Ed., Hillsdale, NJ: Erlbaum Assoc, 1978.

[86] TechTarget, *Refactoring*, 2024. [Online]. Available: https://www.techtarget.com/searchapparchitecture/definition/refactoring (visited on 05/12/2024).

[87] Dan the AI Man, *Unveiling the development roadmap for babyagi*, 2023. [Online]. Available: https://gpt-insight.com/unveiling-the-development-roadmap-for-babyagi/#Decoding_the_Architecture_of_BabyAGI (visited on 05/12/2024).

[88] C. Borek, "Comparative evaluation of llm-based approaches to chatbot creation," *Trepo*, 2024. [Online]. Available: https://trepo.tuni.fi/bitstream/handle/10024/154995/BorekCecylia.pdf?sequence=2.

A

Appendix 1

A.1 Continue code listings

This section includes the code listings from the open-source GitHub repository of the Continue project.

A.1.1 Resolve Input

```
async function resolveEditorContent(
  editorState: JSONContent
): Promise<[ContextItemWithId[], MessageContent]> {
  let parts: MessagePart[] = [];
  let contextItemAttrs: MentionAttrs[] = [];
  const selectedCode: RangeInFile[] = [];
  let slashCommand = undefined;
  for (const p of editorState?.content) {
    if (p.type === "paragraph") {
      const [text, ctxItems, foundSlashCommand] = resolveParagraph(
        p);
      if (foundSlashCommand && typeof slashCommand === "undefined")
        {
        slashCommand = foundSlashCommand;
      }
      if (text === "") {
        continue;
      }

      if (parts[parts.length - 1]?.type === "text") {
        parts[parts.length - 1].text += "\n" + text;
      } else {
        parts.push({ type: "text", text });
      }
      contextItemAttrs.push(...ctxItems);
    } else if (p.type === "codeBlock") {
      if (!p.attrs.item.editing) {
        const text =
          "```" + p.attrs.item.name + "\n" + p.attrs.item.content +
            "\n```";
        if (parts[parts.length - 1]?.type === "text") {
          parts[parts.length - 1].text += "\n" + text;
        } else {
          parts.push({
            type: "text",
```

```
            text,
          });
        }
      }

      const name: string = p.attrs.item.name;
      let lines = name.substring(name.lastIndexOf("(") + 1);
      lines = lines.substring(0, lines.lastIndexOf(")"));
      const [start, end] = lines.split("-");

      selectedCode.push({
        filepath: p.attrs.item.description,
        range: {
          start: { line: parseInt(start) - 1, character: 0 },
          end: { line: parseInt(end) - 1, character: 0 },
        },
      });
    } else if (p.type === "image") {
      parts.push({
        type: "imageUrl",
        imageUrl: {
          url: p.attrs.src,
        },
      });
    } else {
      console.warn("Unexpected content type", p.type);
    }
  }

  let contextItemsText = "";
  let contextItems: ContextItemWithId[] = [];
  const ide = new ExtensionIde();
  for (const item of contextItemAttrs) {
    if (item.itemType === "file") {
      // This is a quick way to resolve @file references
      const basename = getBasename(item.id);
      const content = await ide.readFile(item.id);
      contextItemsText += `\`\`\`title="${basename}"\n${content}\n
        \`\`\`\n`;
      contextItems.push({
        name: basename,
        description: item.id,
        content,
        id: {
          providerTitle: "file",
          itemId: item.id,
        },
      });
    } else {
      const data = {
        name: item.itemType === "contextProvider" ? item.id : item.
          itemType,
        query: item.query,
        fullInput: stripImages(parts),
        selectedCode,
      };
```

```
87        const { items: resolvedItems } = await ideRequest(
88          "getContextItems",
89          data
90        );
91        contextItems.push(...resolvedItems);
92        for (const resolvedItem of resolvedItems) {
93          contextItemsText += resolvedItem.content + "\n\n";
94        }
95      }
96    }
97
98    if (contextItemsText !== "") {
99      contextItemsText += "\n";
100   }
101
102   if (slashCommand) {
103     let lastTextIndex = findLastIndex(parts, (part) => part.type
          === "text");
104     parts[lastTextIndex].text = `${slashCommand} ${parts[
          lastTextIndex].text}`;
105   }
106
107   return [contextItems, parts];
108 }
```

Listing A.1: Resolve Input Typescript file

(https://github.com/continuedev/continue/blob/8ba15b16665be871e037ada88d51b3403a8d094e/gui/src/components/mainInput/resolveInput.ts#L27)

A.1.2 Debug Panel

```
1       const items = await provider.getContextItems(query, {
2             llm,
3             embeddingsProvider: config.embeddingsProvider,
4             fullInput,
5             ide,
6             selectedCode,
7           });
```

Listing A.2: Debug Panel Typescript file

(https://github.com/continuedev/continue/blob/8ba15b16665be871e037ada88d51b3403a8d094e/extensions/vscode/src/debugPanel.ts#L586-L592)

A.1.3 Use Chat Handler

```
1         let newHistory: ChatHistory = [...history.slice(0, index)
            , historyItem];
```

Listing A.3: Use Chat Handler

(https://github.com/continuedev/continue/blob/8ba15b16665be871e037ada88d51b3403a8d094e/gui/src/hooks/useChatHandler.ts#L154)

A.1.4 Count Tokens

```
function compileChatMessages(
  modelName: string,
  msgs: ChatMessage[] | undefined = undefined,
  contextLength: number,
  maxTokens: number,
  supportsImages: boolean,
  prompt: string | undefined = undefined,
  functions: any[] | undefined = undefined,
  systemMessage: string | undefined = undefined
): ChatMessage[] {
  const msgsCopy = msgs ? msgs.map((msg) => ({ ...msg })) : [];

  if (prompt) {
    const promptMsg: ChatMessage = {
      role: "user",
      content: prompt,
    };
    msgsCopy.push(promptMsg);
  }

  if (systemMessage && systemMessage.trim() !== "") {
    const systemChatMsg: ChatMessage = {
      role: "system",
      content: systemMessage,
    };
    // Insert as second to last
    // Later moved to top, but want second-priority to last user message
    msgsCopy.splice(-1, 0, systemChatMsg);
  }

  let functionTokens = 0;
  if (functions) {
    for (const func of functions) {
      functionTokens += countTokens(JSON.stringify(func), modelName);
    }
  }

  if (maxTokens + functionTokens + TOKEN_BUFFER_FOR_SAFETY >= contextLength) {
    throw new Error(
      `maxTokens (${maxTokens}) is too close to contextLength (${contextLength}), which doesn't leave room for response. Try increasing the contextLength parameter of the model in your config.json.`
    );
  }

  // If images not supported, convert MessagePart[] to string
  if (!supportsImages) {
    for (const msg of msgsCopy) {
      if ("content" in msg && Array.isArray(msg.content)) {
        const content = stripImages(msg.content);
```

```
        msg.content = content;
      }
    }
  }

  const history = pruneChatHistory(
    modelName,
    msgsCopy,
    contextLength,
    functionTokens + maxTokens + TOKEN_BUFFER_FOR_SAFETY
  );

  if (
    systemMessage &&
    history.length >= 2 &&
    history[history.length - 2].role === "system"
  ) {
    const movedSystemMessage = history.splice(-2, 1)[0];
    history.unshift(movedSystemMessage);
  }

  const flattenedHistory = flattenMessages(history);

  return flattenedHistory;
}
```

Listing A.4: Count tokens

(https://github.com/continuedev/continue/blob/8ba15b16665be871e037ada88d51b3403a8d094e/core/llm/countTokens.ts#L253)

A.1.5 Chat

```
import { ChatMessage } from "../../index.js";
import { stripImages } from "../countTokens.js";

function templateFactory(
  systemMessage: (msg: ChatMessage) => string,
  userPrompt: string,
  assistantPrompt: string,
  separator: string,
  prefix?: string,
  emptySystemMessage?: string,
): (msgs: ChatMessage[]) => string {
  return (msgs: ChatMessage[]) => {
    let prompt = prefix ?? "";

    // Skip assistant messages at the beginning
    while (msgs.length > 0 && msgs[0].role === "assistant") {
      msgs.shift();
    }

    if (msgs.length > 0 && msgs[0].role === "system") {
      prompt += systemMessage(msgs.shift()!);
    } else if (emptySystemMessage) {
```

```
      prompt += emptySystemMessage;
    }

    for (let i = 0; i < msgs.length; i++) {
      const msg = msgs[i];
      prompt += msg.role === "user" ? userPrompt : assistantPrompt;
      prompt += msg.content;
      if (i < msgs.length - 1) {
        prompt += separator;
      }
    }

    if (msgs.length > 0 && msgs[msgs.length - 1].role === "user") {
      prompt += separator;
      prompt += assistantPrompt;
    }

    return prompt;
  };
}

/**
 * @description Template for LLAMA2 messages:
 *
 * <s>[INST] <<SYS>>
 * {{ system_prompt }}
 * <</SYS>>
 *
 * {{ user_msg_1 }} [/INST] {{ model_answer_1 }} </s><s>[INST] {{
    user_msg_2 }} [/INST] {{ model_answer_2 }} </s><s>[INST] {{
    user_msg_3 }} [/INST]
 */
function llama2TemplateMessages(msgs: ChatMessage[]): string {
  if (msgs.length === 0) {
    return "";
  }

  if (msgs[0].role === "assistant") {
    // These models aren't trained to handle assistant message
       coming first,
    // and typically these are just introduction messages from
       Continue
    msgs.shift();
  }

  let prompt = "";
  let hasSystem = msgs[0].role === "system";

  if (hasSystem && stripImages(msgs[0].content).trim() === "") {
    hasSystem = false;
    msgs = msgs.slice(1);
  }

  if (hasSystem) {
    const systemMessage = `<<SYS>>\n ${msgs[0].content}\n<</SYS>>\n
       \n`;
```

```
    if (msgs.length > 1) {
      prompt += `<s>[INST] ${systemMessage} ${msgs[1].content} [/
          INST]`;
    } else {
      prompt += `[INST] ${systemMessage} [/INST]`;
      return prompt;
    }
  }

  for (let i = hasSystem ? 2 : 0; i < msgs.length; i++) {
    if (msgs[i].role === "user") {
      prompt += `[INST] ${msgs[i].content} [/INST]`;
    } else {
      prompt += msgs[i].content;
      if (i < msgs.length - 1) {
        prompt += "</s>\n<s>";
      }
    }
  }

  return prompt;
}

function anthropicTemplateMessages(messages: ChatMessage[]): string
    {
  const HUMAN_PROMPT = "\n\nHuman:";
  const AI_PROMPT = "\n\nAssistant:";
  let prompt = "";

  // Anthropic prompt must start with a Human turn
  if (
    messages.length > 0 &&
    messages[0].role !== "user" &&
    messages[0].role !== "system"
  ) {
    prompt += `${HUMAN_PROMPT} Hello.`;
  }
  for (const msg of messages) {
    prompt += `${
      msg.role === "user" || msg.role === "system" ? HUMAN_PROMPT :
          AI_PROMPT
    } ${msg.content} `;
  }

  prompt += AI_PROMPT;
  return prompt;
}

`A chat between a curious user and an artificial intelligence
    assistant. The assistant gives helpful, detailed, and polite
    answers to the user's questions.
USER: <image>{prompt}
ASSISTANT:`;
const llavaTemplateMessages = templateFactory(
  () => "",
  "USER: <image>",
```

```
  "ASSISTANT: ",
  "\n",
  "A chat between a curious user and an artificial intelligence
     assistant. The assistant gives helpful, detailed, and polite
     answers to the user's questions.",
);

const zephyrTemplateMessages = templateFactory(
  (msg) => `<|system|>${msg.content}</s>\n`,
  "<|user|>\n",
  "<|assistant|>\n",
  "</s>\n",
  undefined,
  "<|system|> </s>\n",
);

const chatmlTemplateMessages = templateFactory(
  (msg) => `<|im_start|>${msg.role}\n${msg.content}<|im_end|>\n`,
  "<|im_start|>user\n",
  "<|im_start|>assistant\n",
  "<|im_end|>\n",
);

const templateAlpacaMessages = templateFactory(
  (msg) => `${msg.content}\n\n`,
  "### Instruction:\n",
  "### Response:\n",
  "\n\n",
  undefined,
  "Below is an instruction that describes a task. Write a response
     that appropriately completes the request.\n\n",
);

function deepseekTemplateMessages(msgs: ChatMessage[]): string {
  let prompt = "";
  let system: string | null = null;
  prompt +=
    "You are an AI programming assistant, utilizing the DeepSeek
       Coder model, developed by DeepSeek Company, and you only
       answer questions related to computer science. For
       politically sensitive questions, security and privacy issues
       , and other non-computer science questions, you will refuse
       to answer.\n";
  if (msgs[0].role === "system") {
    system = stripImages(msgs.shift()!.content);
  }

  for (let i = 0; i < msgs.length; i++) {
    const msg = msgs[i];
    prompt += msg.role === "user" ? "### Instruction:\n" : "###
       Response:\n";

    if (system && msg.role === "user" && i === msgs.length - 1) {
      prompt += `${system}\n`;
    }

```

```
    prompt += `${msg.content}`;

    if (i < msgs.length - 1) {
      prompt += msg.role === "user" ? "\n" : "<|EOT|>\n";
    }
  }

  if (msgs.length > 0 && msgs[msgs.length - 1].role === "user") {
    prompt += "\n";
    prompt += "### Response:\n";
  }

  return prompt;
}

// See https://huggingface.co/microsoft/phi-2#qa-format
const phi2TemplateMessages = templateFactory(
  (msg) => `\n\nInstruct: ${msg.content} `,
  "\n\nInstruct: ",
  "\n\nOutput: ",
  " ",
);

const phindTemplateMessages = templateFactory(
  (msg) => `### System Prompt\n${msg.content}\n\n`,
  "### User Message\n",
  "### Assistant\n",
  "\n",
);

/**
 * OpenChat Template, used by CodeNinja
 * GPT4 Correct User: Hello<|end_of_turn|>GPT4 Correct Assistant:
    Hi<|end_of_turn|>GPT4 Correct User: How are you today?<|
    end_of_turn|>GPT4 Correct Assistant:
 */
const openchatTemplateMessages = templateFactory(
  () => "",
  "GPT4 Correct User: ",
  "GPT4 Correct Assistant: ",
  "<|end_of_turn|>",
);

/**
 * Chat template used by https://huggingface.co/TheBloke/XwinCoder
    -13B-GPTQ
 *

<system>: You are an AI coding assistant that helps people with
   programming. Write a response that appropriately completes the
   user's request.
<user>: {prompt}
<AI>:
 */
const xWinCoderTemplateMessages = templateFactory(
  (msg) => `<system>: ${msg.content}`,
```

```
  "\n<user>: ",
  "\n<AI>: ",
  "",
  undefined,
  "<system>: You are an AI coding assistant that helps people with programming. Write a response that appropriately completes the user's request.",
);

/**
 * NeuralChat Template
 * ### System:\n{system_input}\n### User:\n{user_input}\n### Assistant:\n
 */
const neuralChatTemplateMessages = templateFactory(
  (msg) => `### System:\n${msg.content}\n`,
  "### User:\n",
  "### Assistant:\n",
  "\n",
);

/**
'<s>Source: system\n\n System prompt <step> Source: user\n\n First user query <step> Source: assistant\n\n Model response to first query <step> Source: user\n\n Second user query <step> Source: assistant\nDestination: user\n\n '
 */
function codeLlama70bTemplateMessages(msgs: ChatMessage[]): string {
  let prompt = "<s>";

  for (const msg of msgs) {
    prompt += `Source: ${msg.role}\n\n ${stripImages(msg.content).trim()}`;
    prompt += " <step> ";
  }

  prompt += "Source: assistant\nDestination: user\n\n";

  return prompt;
}

const llama3TemplateMessages = templateFactory(
  (msg: ChatMessage) =>
    `<|begin_of_text|><|start_header_id|>${msg.role}<|end_header_id|>\n${msg.content}<|eot_id|>\n`,
  "<|start_header_id|>user<|end_header_id|>\n",
  "<|start_header_id|>assistant<|end_header_id|>\n",
  "<|eot_id|>",
);

/**
 <start_of_turn>user
 What is Cramer's Rule?<end_of_turn>
 <start_of_turn>model
 */
```

```
const gemmaTemplateMessage = templateFactory(
  () => "",
  "<start_of_turn>user\n",
  "<start_of_turn>model\n",
  "<end_of_turn>\n",
);

export {
  anthropicTemplateMessages,
  chatmlTemplateMessages,
  codeLlama70bTemplateMessages,
  deepseekTemplateMessages,
  gemmaTemplateMessage,
  llama2TemplateMessages,
  llama3TemplateMessages,
  llavaTemplateMessages,
  neuralChatTemplateMessages,
  openchatTemplateMessages,
  phi2TemplateMessages,
  phindTemplateMessages,
  templateAlpacaMessages,
  xWinCoderTemplateMessages,
  zephyrTemplateMessages,
};
```

Listing A.5: Chat

(https://github.com/continuedev/continue/blob/main/core/llm/templates/chat.ts)

A.1.6 Adapter

```
    for await (const chunk of this._streamComplete(prompt,
        completionOptions)) {
      completion += chunk;
      yield chunk;
    }
```

Listing A.6: Adapter

(https://github.com/continuedev/continue/blob/8ba15b16665be871e037ada88d51b3403a8d094e/core/llm/index.ts#L283)

A.2 Interviews

This part of the appendix is for showing the conducted interviews with developers. The participants of the following interviews have consented to having them be included. Interviews where that is not the case at the time of writing have been excluded from this section.

A.2.1 Included projects

A.2.1.1 Continue

This interview was conducted with one of the founders of Continue. Some hyperlinks are unavailable here, however they are properly referenced in the result section.

Interviewer:

> I'm currently researching some things relating to applications using LLM's, and I'm wondering if there may be a graph, or similar, showing how continue works somewhat abstractly. Or maybe some document describing the flow or abstract structure?

Interviewee:

> Can you tell me more about what specifically you're researching? There are many different levels of abstraction that we have that show how Continue works, so it would be helpful to know what level(s) are most interesting to you.

Interviewer:

> It's specifically within the area of architectures and design patterns, so like the main "actions" or "processes" taking place from start to finish between the main modules during one process loop. For example, this is how I described promptify (even if there is some lacking details), another subject in my research.

Interviewee:

> Here's basically what happens on each input, lmk if you want me to share links to any of the code
>
> 1. User input (can include code blocks, images, "@" references, "slash commands")
> 2. This multi-media input is converted into a common JSON format of just raw text or base64 images. As part of this, all "@" references have functions that get called, sometimes performing RAG, other times retrieving information like the git diff, etc... and it is added to the top of the prompt
> 3. This prompt is added to all of the previous messages from the conversation
> 4. We take into account the context length of the model in order to prune the input so that it fits
> 5. If the API being used doesn't automatically do prompt formatting, we format for whichever model is being used
> 6. Send to the LLM API
> 7. Receive stream and display in UI
>
> For our diff streaming feature we add in a lot of post-processing.

Interviewer:

> Sorry to bother you with this again, but there is some more information that I would need regarding this. Specifically what entities or modules, or similar, perform these actions, I really can't make my chart without that.

Interviewee:

When you say entities, do you mean something similar to what you have in your chart above (e.g. Prompter, Pipeline, Logger, etc...)? We don't necessarily have names for anything like this, it's all just a handful of different functions in the same codebase, though I would gladly make them up. Or do you mean something different?

Interviewer:

Either that or what function does what. In the example chart, the pipeline is basically just the function that is most central, and when it sends and receives stuff from the prompter that is just the function calling a function in the prompter, so it's essentially the flow of the main/most important function. But if your code isn't really structured like that, then making up names for "entities" could work.

Interviewee:

Here are some links to functions and their made up names for the points mentioned before:

1. User input (can include code blocks, images, "@" references, "slash commands")
2. "Serializer": This multi-media input is converted into a common JSON format of just raw text or base64 images. "Context Providers": As part of this, all "@" references have functions that get called, sometimes performing RAG, other times retrieving information like the git diff, etc... and it is added to the top of the prompt
3. "Chat Handler": This prompt is added to all of the previous messages from the conversation (in useChatHandler
4. "Pruner": We take into account the context length of the model in order to prune the input so that it fits (`https://github.com/continuedev/continue/blob/8ba15b16665be871e037ada88d51b3403a8d094e/core/llm/countTokens.ts#L253`)
5. "Formatter": If the API being used doesn't automatically do prompt formatting, we format for whichever model is being used (here are all of the templates)
6. "LLM Adapter": Send to the LLM API (`https://github.com/continuedev/continue/blob/8ba15b16665be871e037ada88d51b3403a8d094e/core/llm/index.ts#L283`). This is the abstraction that lets us send to any LLM API
7. Receive stream and display in UI So a summary with these names I've come up with: Multi-media input -> "Serializer" -> "Context Providers" -> "Chat Handler" -> "Pruner" -> "Formatter" -> "LLM Adapter" -> Display

Let me know if this is more helpful!

A.2.1.2 InvokeAI

This interview was not conducted with a developer from the project but a contributor.

Interviewer:

where should I ask about how Invoke works internally?

Interviewee:

dev-chat or any of the more specific channels in the dev sections. You can also look through the codebase on GitHub if you're already versed in python and typescript

Interviewer:

it's for research purposes, I don't really have the time to analyse codebases in detail I would like to visualise the abstract flow of the code and I found the chart in the architecture section quite lacking this one `https://invoke-ai.github.io/InvokeAI/contributing/ARCHITECTURE/#local-development`

Interviewee:

What is the target of your research? We are a node-based frontend for diffusers. We have customizations on it, but if your research is on the AI inference itself, you'll be better off looking at Huggingface's Diffusers code.

Interviewer:

well, I'm researching the structure of applications using AI, as opposed to the AI itself, so I'm looking for the flow of calls and information from input to output (where the AI is merely one "entity"), so that I can make a flow chart out of it, basically looking for general abstract structures through that

Interviewee:

I see. Well, I normally keep my head in the low-level inference side of things, but I can give you a basic overview.

- The frontend is a webserver built on React and Typescript, and the backend runs in Python as most AI software does at this point since they all use Torch library.
- Our backend is a node-based architecture, where modular processing steps are linked together in a graph to form the full pipeline.
- When you invoke a basic inference pipline from the frontend (txt2img, for instance), it builds a graph from a pre-selected format based on your inputs. The initial values in the graph are filled from the settings in the web UI, and some nodes in the graph may be inserted/removed based on what settings are enabled.
- The graph is sent through an API to the backend where it is put in a queue behind any previous or ongoing processes.
- Once the graph reaches execution, it calls the invoke() method of each node and passes the output values to the input values of the next node in the sequence.
- Usually the last stage is to have a node that saves an image output from one of the previous nodes, but not always.

Some special considerations:

- Rather than using a pre-selected graph format, there is also a Workflow Editor that allows users to structure their own node graph for better control. For examples of what those look like, check out share-your-workflows

- Outputs from nodes are stored in a database or to disk, and caching is enabled such that if a node is executed in multiple graphs with all of the same inputs, it will skip the processing and use the cached result instead.
- If a graph fails from some internal error, then the local process stops and moves on to the next graph in the queue.

That's about as specific as I can get for a lot of that. Other devs might chime in and correct or clarify where I've gone wrong. Good luck on your midterm paper

A.2.1.3 GPT Pilot

The interviewee here seemed to be a developer, their discord tags included "Pythagora Team".

Interviewer:

hello, where should I post if I have some research questions regarded GPT Pilot?

Interviewee:

Hey [Discord handle], welcome! Here's a good place, ask away! Also check out our blog at `https://blog.pythagora.ai/` (not a lot of content there but we've got a few articles going in-depth in Pythagora design)

Interviewer:

I am a Master's student in Computer Science, currently researching some things relating to applications and projects using LLMs, and I'm wondering if there may be a graph, or similar, showing how GPT Pilot works, in an abstract fashion. Or maybe some document describing the flow or abstract structure? This is research about finding possible design patterns and architectures for said LLM based applications and projects. If there aren't any visuals, like graphs, available, would it be possible for a description of it from which I could draw something up?

Interviewee:

did you check the blog posts I mentioned?

Interviewer:

Yes, I did, the only somewhat relevant post I found was this one: `https://blog.pythagora.ai/2023/09/04/gpt-pilot-coding-workflow-part-2-3/` And its previous part They did help establish that GPT pilot uses a multi-agent framework. But I would like to know, if possible, how each agent is built. Like what different internal parts there are, and how the information flow is for an execution loop, as in input to output.

Interviewee:

We don't have more documentation around that but since the code is open source you could dive in here: `https://github.com/Pythagora-io/gpt-pilot/tree/main/pilot/helpers/agents` (the code is a bit of a mess and actually some of the parts are incorrectly implemented as functions within Developer, but source code is the ultimate truth there)

A.2.2 Disqualified projects

A.2.2.1 GPT-Engineer

The interviewee for this project was a moderator and contributor. This project was disqualified for having little to no useful information.

Interviewer:

> Hello I'm currently researching some things relating to applications and projects using LLM's, and I'm wondering if there may be a graph, or similar, showing how gpt-engineer works somewhat abstractly. Or maybe some document describing the flow or abstract structure? This is research about finding possible design patterns and architectures for said LLM based applications projects. If there aren't any visuals, like graphs, available, would it be possible for a description of it from which I could draw something up? I'm also using this graph as the comparator, to see if there are any similarities to it: `https://github.blog/wp-content/uploads/2023/10/LLMapparchitecturediagram.png?resize=1600%2C850`

Interviewee:

> Hey [Discord handle] I'll ask about this at our team meeting today and get back to you!

Interviewee:

> Not sure if this is helpful for you to get some level of understanding our internal structure too? `https://gpt-engineer.readthedocs.io/en/latest/introduction.html` We have this too `https://gpt-engineer.readthedocs.io/en/stable/intro/gate.html` It's an older version though!

Interviewer:

> unfortunately this is not that helpful to my research that one [the second link] is a bit more helpful, but it still lacks some detail, namely how, or if, it communicates with the LLM, if it has some prompt processing steps or not, but the component list is good to have in mind

Interviewee:

> Fair point. I'll ask at our team meeting. It's open to you if you want to join as well. It gets pretty technical but in any case I'll ask about any resources, class diagrams or schematics we might have.

Interviewer:

> I wouldn't want to intrude, but I can provide examples of other things I've drawn up while researching other projects basically the information flow from input to output [attached images]

Interviewee:

> Thanks for this! If you don't mind me asking, what's your background? We are looking very much into extending our relationship with the academic and research sector, and I love to learn about people using and researching what we are building upon.

Interviewer:

I'm not that high up I'm afraid, this is for a master's thesis, graduation work essentially. It's for a master's degree within software engineering.

Interviewee:

That's perfectly fine. I'll see what I can do for you! Where are you located?

Interviewer:

Sweden, studying at the Chalmers University of Technology

Interviewee:

Oh wow, most of the original team is from Sweden too!

Interviewer:

oh, what a coincidence, haha

Interviewee:

Yup! In any case, I'll check with the team and let you know. Good luck with your future work!

Interviewer:

Thank you!

Interviewer:

[Four days later] Hi, don't want to rush you or anything, just wondering if you have any news for me?

Interviewee:

Hey, I do have some news - I wanted to dig into our docs and see if we can do anything, but we don't have specific schematics. One thing that comes to mind is that there are probably AI or other tools that can look into the repo and make a class diagram or something of the kind. I'm sorry I don't have a much better solution for you right now

Interviewer:

It's fine. Besides, I don't actually need visuals, I can draw those myself. I just need descriptions I can work off of. But if you don't have those either, I guess I'll have to figure something out, with the worst case scenario being I can't include it in the thesis

Interviewee:

Sadly nothing specific. But try looking for some class diagram generators or code explainers, maybe that'll yield some results.

Interviewer:

you know any examples of such? I'll try looking for something, thanks for being so helpful and interested, not a lot of devs are like that

Interviewee:

I'm sorry I'm running between meetings and user interviews, I'll get back to you a little bit later today!

Interviewee:

[Two days later] Sorry, I was getting swamped. I don't have a specific tool to recommend - maybe ask Chat GPT-4 or Google for Python class/repo visualizer? I'm really not sure what's the best option here.

Interviewer:

It's fine I did look for some, but unfortunately most were unavailable for some reason thanks anyway for your help

Interviewee:

> Of course, sorry it wasn't more useful. I wish you good luck with your thesis!

Since no such tool was available, the project was disqualified.

A.2.2.2 PR-Agent

The developers of PR-Agent were helpful at first, but when asked for more detailed flow charts, they said that they didn't have it and implied that they would not have shared it even if they did have it. The interviewees were the CEO of CodiumAI and an engineer.

Interviewer:

> Hello I'm currently researching some things relating to applications using LLM's, and I'm wondering if there may be a graph, or similar, showing how pr-agent works somewhat abstractly. Or maybe some document describing the flow or abstract structure? Another way of putting it is that I'm looking for an abstract overview of how the information flow works during one execution loop.

Interviewee 1:

> Try: `https://github.com/Codium-ai/pr-agent?tab=readme-ov-file#how-it-works` `https://pr-agent-docs.codium.ai/`

Interviewer:

> thank you, but that chart is missing some things, like how the messages to the LLM work, if there are components such as an embedding model, vector database, and a data filter. My research group is using this as the comparator when researching [Attached Figure 3.3 image]

Interviewee 2:

> [Discord handle] it's not a bad idea to make this kind of flow chart, but currently we dont have one for PR-Agent.
>
> For AlphaCodium we do have something more similar:
>
> `https://github.com/Codium-ai/AlphaCodium` As a side note, notice that the flow of "real" products are usually quite different than the flow presented in research projects.
>
> "real" products need to work fast (in real-time, or close to it), and have as few LLM calls as possible, since it costs a lot of money.
>
> So usually the flow chart for "real" products will be simpler than the one you portrayed

Interviewer:

> When there isn't a chart available I usually make it myself based on developer descriptions. Which is kind of what I'm asking for here. And that chart doesn't really work for my research either since it just shows the actions, when I also need to know what entities perform said actions (even if those entities are just informal).
>
> When there isn't a chart available I usually make it myself based on developer descriptions. Which is kind of what I'm asking for here. [Referring to the second part of the statement] And that chart doesn't really work for my research either since it just shows the actions, when I also

need to know what entities perform said actions (even if those entities are just informal).

Interviewee:

We do not have an existing chart for PR-Agent to provide, sorry. Also note that it is usually a sensitive commercial data. A lot of open-source projects (including ours) have a commercial branch. So an accurate flow chart, including all the aspects you mentioned (logs, telemetry, authentications, database, filtering, ...) is not often shared

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY