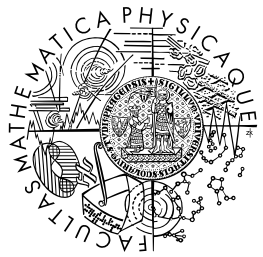

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Michal Pospěch

MetaGraph: Constructing graph-based agents through meta-programming

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	Mgr. Tomáš Petříček, Ph.D.
Study programme:	Computer Science
Study branch:	Artificial Intelligence

Prague 2025

I declare that I carried out this master thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I'd like to thank my friends, family and colleagues for their support over the years. And I'd also like to thank my supervisor Mgr. Tomáš Petříček, Ph.D. for his guidance and patience.

Title: MetaGraph: Constructing graph-based agents through meta-programming

Author: Bc. Michal Pospěch

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Tomáš Petříček, Ph.D., Department of Distributed and Dependable Systems

Abstract: There are many frameworks for creation of LLM-based agents. However, their API tends to be hard to work with. This thesis builds upon the idea that the best way how to describe a workflow is code and introduces the MetaGraph metaprogramming framework. The framework creates LangGraph agents by parsing Python code and transforming it into a graph. It is shown that while this approach is completely valid, it is suited mostly for workflows that are akin to procedural programming, while workflows that are non-linear in nature are harder to model and require goto-like constructs.

Keywords: meta-programming Agentic AI LangGraph LLM

Contents

Introduction **3**

1 Large Language Models **5**
1.1 History 5
1.1.1 Transformer-based Neural Language Models 5
1.1.2 Large Language Models 7
1.2 Architecture and how LLMs are made 8
1.2.1 Transfomer-Based Architecture 8
1.2.2 Data quality 10
1.2.3 Tokenisation 10
1.2.4 Positional Encoding 11
1.2.5 Pre-training 11
1.2.6 Fine-tuning 11
1.2.7 Alignment 12
1.2.8 Decoding Samples 12
1.3 LLM Usage 12
1.3.1 Limitations 12
1.3.2 Prompt Engineering 14
1.3.3 Retrieval Augmented Generation 15
1.3.4 Using External Tools 15

2 Agents **17**
2.1 Agents from the Philosophical Perspective 17
2.2 Technology in AI agents in time 17
2.3 What are agents? 19
2.3.1 Perception 19
2.3.2 Brain 20
2.3.3 Actions 23

3 Metaprogramming **25**
3.1 Models . . . 25
3.1.1 Macro Systems . . . 25
3.1.2 Reflection Models . . . 26
3.1.3 Metaobject Protocols . . . 26
3.1.4 Aspect-Oriented Programming . . . 26
3.1.5 Generative Programming . . . 27
3.1.6 Multistage Programming . . . 28
3.2 Phase of Evaluation . . . 28
3.2.1 Preprocessing-Time Evaluation . . . 28
3.2.2 Compilation-time Evaluation . . . 29
3.2.3 Execution-time Evaluation . . . 29
3.3 Metaprogram Source Location . . . 29
3.3.1 Embedded in the Subject Program . . . 29
3.3.2 External to the Subject Program . . . 30

4 Frameworks **31**
4.1 LangChain Platform . . . 32
4.1.1 LangChain . . . 33
4.1.2 LangGraph . . . 34

5 MetaGraph **41**
5.1 Motivation . . . 41
5.2 Design . . . 41
5.3 Analysis . . . 46
5.4 Implementation . . . 47
5.4.1 Grammar . . . 48
5.4.2 Implementation Details . . . 49
5.5 Showcase . . . 50
5.5.1 Retrieval Augmented Generation . . . 51
5.5.2 Open Deep Research . . . 51
5.6 Limitations . . . 57

Conclusion **59**

Bibliography **61**

Introduction

Large language models have been getting a lot of attention ever since the release of GPT-3. One of the many applications of LLMs are agents. These largely expand the capabilities of LLMs with memory, tools, additional inputs, etc. However in its core it's still software. There are many framworks to make creating this type of software easier, one of them being LangGraph. It enables to model agentic workflows as graph. This however comes with API that is unwieldy. Since everything is split into nodes and edges and control flow is in multiple places it is hard to orient in. One example being Open Deep Research implementation by LangChain, see 1.

In this thesis a metaprogramming system MetaGraph is proposed. It enables the creation of graph-based agents without the explicit construction of a graph. It instead transforms Python code into a graph. This makes it easier for developers to create agents with LangGraph without having to learn any new tools or APIs.

In the first chapter a brief overview of LLMs, their history and how they are built, is provided. This is followed by an overview of agents, first their history within AI along with a more philosophical definition. Following that is an outline of what are the components of an agent. Then metaprogramming is introduced along with classification of metaprogramming systems by way of operation, code location and phase of evaluation. After that frameworks for easier creation of LLM-agents are evaluated and LangChain along with LangGraph are described in greater detail. In the last chapter MetaGraph is presented, providing the core algorithm, analysis of considered approaches, implementation details and a showcase.

```
section_builder = StateGraph(SectionState,
 ↪  output=SectionOutputState)
section_builder.add_node("generate_queries", generate_queries)
section_builder.add_node("search_web", search_web)
section_builder.add_node("write_section", write_section)

# Add edges
section_builder.add_edge(START, "generate_queries")
section_builder.add_edge("generate_queries", "search_web")
section_builder.add_edge("search_web", "write_section")


# Add nodes
builder = StateGraph(ReportState, input=ReportStateInput,
 ↪  output=ReportStateOutput, config_schema=Configuration)
builder.add_node("generate_report_plan", generate_report_plan)
builder.add_node("human_feedback", human_feedback)
builder.add_node("build_section_with_web_research",
 ↪  section_builder.compile())
builder.add_node("gather_completed_sections",
 ↪  gather_completed_sections)
builder.add_node("write_final_sections", write_final_sections)
builder.add_node("compile_final_report", compile_final_report)

# Add edges
builder.add_edge(START, "generate_report_plan")
builder.add_edge("generate_report_plan", "human_feedback")
builder.add_edge("build_section_with_web_research",
 ↪  "gather_completed_sections")
builder.add_conditional_edges("gather_completed_sections",
 ↪  initiate_final_section_writing, ["write_final_sections"])
builder.add_edge("write_final_sections",
 ↪  "compile_final_report")
builder.add_edge("compile_final_report", END)

graph = builder.compile()
```

Listing 1 Example of LangGraph code

Chapter 1

Large Language Models

This chapter focuses on both history of LLMs and on how they are constructed. Its content and structure are inspired by [1].

1.1 History

Language modeling has been a topic of research since the 1950's with Shannon measuring how well an n-gram language predicts or compresses natural languages.[2]

Language modeling using neural networks (NLMs - Neural language models) dates back to the 1980's[3]. These were first based on recurent neural networks (RNNs) or their variations such as LSTM or GRU. They were utilised mostly for machine translation, text generation and text classification.[1]

1.1.1 Transformer-based Neural Language Models

The big breakthrough in development of NLMs was the Transformer architecture. [4] It applies self-attention in parallel for every word in a sentence to compute an "attention score" a measure of how do words in a sentence influence each other. They enable more parallelisation than RNNs making them easier to pre-train efficiently on GPUs. These Pre-Trained Language Models (PLMs) can then be fine-tuned to various downstream tasks. Early Transformer-based PLMs can be divided into 3 distinct categories.[1]

Encoder-only models First there are encoder-only PLMs. These contain only the encoder network. Originally they were developed for langauge understanding tasks such as text classification. The most representative model is **BERT** [5] and its derivatives some of wich will be outlined below. BERT (Birectional Encoder

Representations from Transformers) consists of three modules: embedding module transforming textual input into a sequence of embedding vectors, stack of Transformer encoders transforming the embeddings into contextual representation vecors and a fully connected layer converting the representations into one-hot vectors. BERT is pretrained using 2 objectives, next sentence prediction and masked language modelling. The pretrained model can then be fine-tuned for various tasks by adding a classifier layer. BERTs success lead to many more encoder-only architectures being developed. **RoBERTA** [6] improved the robustness of BERT, **ALBERT** [7] decreased memory consumption, **DeBERTa** [8] used modified attention mechanism and enhanced mask-decoder. **ELECTRA** [9] used different pre-training task (replaced token prediction). XLMs (cross-lingual language models) [10] extended BERT to work with multiple languages. Other models such XLNet[11] or UNILM[12] solely use auto-regressive models for training and inference.

Decoder-only models Next there are decoder-only models. The most prominent are **GPT-1**[13] and **GPT-2**[14] which laid the foundations for GPT-3 and GPT-4. GPT-1 was the first to show that generative pre-training of decoder-only architectures on a diverse dataset in a self-supervised manner can reach good performance in wide range of natural language tasks. This is followed by discriminative fine-tuning for specific tasks. GPT-2 then further improves upon GPT-1 by changing the architecture slightly (different normalization, vocabulary size increased and context size increased) and shows that by training on large WebText dataset without any explicit supervision models can learn to perform various natural language tasks.

Encoder-decoder models As last there are encoder-decoder models. These can model almost all NLP tasks since almost all such tasks can be expressed as sequence-to-sequence text generation tasks[15]. The most representative models from this group are **T5**[15], **mT5**[16], **MASS**[17] and **BART**[18]. In **T5** and **mT5**, its multilingual variant, a unified fraamework is proposed which casts all NLP tasks as text-to-text generation task. First a model is trained on a massive corpus of data and then it is finetuned for specific NLP tasks. In MASS encoder taks sequnece of tokens where a fragment is masked and the decoder's task is to predict it. This way both the decoder and encoder are trained for language embedding and generation, respecitvely, at the same time. BART is pretrained by introducing some noise into the input sequences and learning how to reconstruct it.

1.1.2 Large Language Models

Large language models mainly refer to transformer-based PLMs with large amount of parameters (billions). When compared to aforementioned PLMs they are not only much bigger but also show significantly better language understanding and generation capabilities. There are 3 distinct families. GPT, LLaMA and PaLM.

GPTs are a family of Transformer-based decoder-only models created by OpenAI. This family consists of **GPT-1**, **GPT-2**, **GPT-3**[19], **InstructGPT**[20], **ChatGPT**[21], **GPT-4**[22], **CODEX**[23] and **WebGPT**[24]. Early models (GPT-1 and 2) are open-source, the rest is not and is only available via an API. GPT-3 released in 2020 is considered the first LLM not only because of its size (175B parameters) but also demonstrates capabilities not observed in previous models. It is a pre-trained autoregressive model. The model shows the capability of in-context learning meaning that it can be applied to any downstream task without the need to update gradients or do fine-tuning. Providing textual descriptions or exammples is enough. CODEX released in 2023 is a descendant of GPT-3 fine-tuned for programming applications using data from GitHub. It currently powers Microsoft GitHub Copilot. WebGPT is also derived from GPT-3 designed to answer open-ended quetion using a web browser. Its training has 3 steps, learning to mimic human web browsing patterns from data. Then learning a reward function to predict human preferences. Finally the reward function is optimized via reinforement learning and rejection sampling. InstructGPT is fine-tuned using reinforcement learning from human feedback to follow expected human instructions. The biggest milestone in development was the introduction of ChatGPT in 2022. It enables the user to steer converstaion to complete various tasks. ChatGPT was powered by GPT-3.5 (sibling to InstructGPT). GPT-4 launched in 2024 is a multimodal model the latest major itreation in the GPT family. The model shows human level-performance in various academic and professional benchmarks[22]. GPT-4 was trained using large text corpora and RLHF (reinforcement learning from human feedback).

LLaMA[25] is a collection of models developed by Meta. These are, unlike GPTs, open-source. Meaning that both the architecture and model weights are available to the research community under a non-commercial license. LLaMA uses Transformer architecture of GPT-3 with modified activation function, embeddings and layer normalization. The LLaMA-13B model outperforms GPT-4 on most benchmark making it good starting point for LLM research. **LLaMA-2**[26], researched in 2023 in partnership with Microsoft, contained **LLaMA-2 Chat**. It was trained by first pre-training LLaMA-2 on publicly available data and then iteratively refined via RLHF. **LLaMA-3** [27] was released in 2024 and offers performence similar to GPT-4 on a plethora of tasks. Other variations of LLaMA include **Alpaca**[28], **Vicuna**[29], **Guanaco**[30], **Koala**[31] or **Mistral-7B**[32],

Code LLaMA[33], **Gorilla**[34], **Giraffe**[35] and many others.

PaLM[36] is a group of models developed by Google. The first was announced in 2022. It's a 54OB parameter transformer based LLM. It was trained on TPUs utilising the Pathways[37] system enabling highly efficient training across multiple TPU pods. It achieved state of the art few-shot learning results on hundreds of language understanding and generation benchmarks and outpeformend not only other models but also humans in some benchmarks. The **U-PaLM**[38] models are trained on PaLM using the UL2R method (a method of continuous training of LLMs) and report circa 50% computational savings. These models were then instruction-finetuned into **Flan-PaLM**[39]. Its method of finetuning used much more tasks, larger models and thought data then works before it. It resulted into it substantially outperforming previous instruction-following models. **PaLM-2**[40] is more compute efficient and has better reasoning and multilingual capabilities compared to PaLM. PaLM-2 is trained using a mixture of objectives. **Med-PaLM**[41] and **Med-PaLM 2**[42] are domain-specific PaLMs aimed at providing high quality answers in the medical field. The latter of these has scored up to 86.5% on MedQA[43] dataset.

1.2 Architecture and how LLMs are made

This section will introduce the most dominant LLM architectures and then dive deeper into various factors of LLM training.

1.2.1 Transfomer-Based Architecture

As mentioned in the previous section, LLMs can be divided into 3 distinct groups based on architecture. Encoder-only, decoder-only and encoder-decoder. All of these draw from *Transformer architecture*. First the Transformer will be outlined and then each of the three architectures will be described.

The Transformer framework was originally designed for effective parallel computing on GPUs. The core idea of it is the (self-)attention mechanism which is able to capture long-term contextual information. The architecture, which was originally developed for the purposes of machine translation, cosists of encoder and decoder. Encoder is a stack of 6 identical Transformer layers. Each of those has 2 sub-layers. First, there is a multi-head self-attention layer. The other one is a simple position-wise fully connected feed-forward network. The decoder is comprised of a stack of N layers (6 in the original work) as well with each having 3 sub-layers compared to the 2 in encoder. The extra layers performs multi-head attention over the output of the encoder stack. The attention function maps a query and set of key-value pairs to an output with query, keys and values being

vectors. Output is calculated as weighted sum of the values with weight being calculated based on compatibility of the query with corresponding key. However instead of using a single attention function with vectors of dimension d_{model} it is beneficial to linearly project the vectors with different linear projections with d_q, d_k and d_l dimensions. To provide information about the positions of the tokens, positional encoding is used.

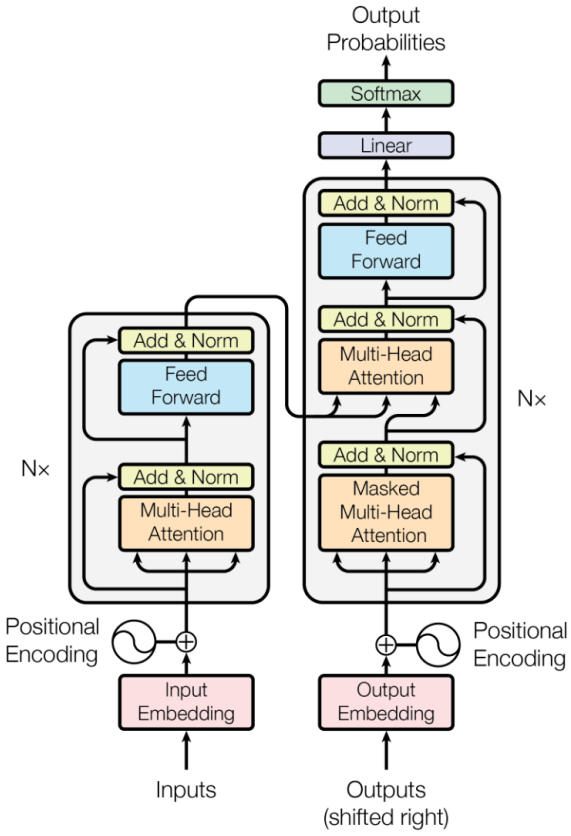

Figure 1.1 Illustration of the Transformer architecture courtesy of [4]

In encoder-only architectures all the attention layers can access the whole initial sentence. They are usually trained by introducing some error into the sentence (e.g. masking a word) and task of the model is to reconstruct the original sentence. They are best used of tasks where understanding the whole sentence is beneficial such as sentence classification. Most prominent example is BERT[5].

In decoder-only models all for any word the attention layers have access to the words preceding it. They are also sometimes call auto-regressive. The task that is usually used to pretrain these models is formulated as predicting the next

token in the sequence. They are best suited for text generation. GPTs are one of the most prominent examples.

Encoder-decoder models, sometimes called sequence-to-sequence models, combine the previous two approaches. Attention layers in the encoder can access the whole sentence while the attention layers in decoder can only access preceding tokens in the input. They are usually trained by an objective for training either encoder or decoder model but with some added complexity (e.g. replacing random spans of texts with a maks word and the goal is to predicted). They re best used for task about generating text conditioned on a given input.

1.2.2 Data quality

Data quality is a important factor influencing perfomrance of LLMs trained on them. Therefore data is cleaned and deduplicated before it's used for training. *Data cleaning* consists of removing noise (e.g. removing false information), handling outliers, balancing the dataset, preprocessing text (removing stop words, punctuation and other elements that do not contribute to the model's learning) and resolving ambiguities. *Deduplication* is needed because a repeated occurence of one sample can introduce bias or lead to overfitting on that particular example. This is particulary important in NLP tasks which require diverse and representative training data. Specific de-duplication methods vary task to task.

1.2.3 Tokenisation

Tokenisation is the process of converting a sequence of text into smaller parts, tokens. Simplest method is to split the text using whitespaces. Most tokenisation methods however rely on word dictionaries. Their problem is out-of-vocabulary (OOV) since the tokeniser simply knows only words in its dictionary. Thus popular tokenisers rely on subwords instead of words to increase coverage of dictionaries. *BytePairEncoding*[44] is originally a data compression algorithm using frequent byte-level patterns to compress the data. Using this algorithm frequent words are kept whole while lest frequent are split. This way the vocabulary is not too large but expressive enough. Also suffixes and prefiexes can be used to model morphological forms of words. *WordPieceEncoding*[45] is used in well known models such as BERT and Electra and shares some basic ideas with BytePairEncoding however the way it creates the vocabulary in a slightly different manner. Unlike these two tokenizers, *SentencePieceEncoding* does not rely on whitespaces to separate words (this is the case for various asian languages)[46].

1.2.4 Positional Encoding

To encode position of token in the original sequence positional encoding is needed. The original Transformer architecture uses *Absolute Positional Embedding*[4]. It attaches positional information to the input embedding at the bottom of both encoder and decoder stacks. They can be both learned or fixed. *Relative Positional Embeddings*[47] extend self-attention to work with links between elements of the input. It is added as a additional component to the keys and a sub-component of the values matrix. It views the input as a graph with weighted directed edges. *Rotary Position Embeddings*[48] improve upon both previous approaches. It uses a rotation matrix to encode the absolute position and includes relative position details in the self-attention mechanism. *Relative Positional Bias*[49] introduces a bias to the attention scores of query-key pairs proportional to their distance.

1.2.5 Pre-training

Pre-training is the first step in the training pipelines for LLMs. It is trained on a massive amount of (usually) unlabeled data in a self-supervised manner. There are multiple approaches. *Autoregressive Language Modelling* tries to predict the next token (or sequence of tokens) using the preceding tokens. Popular loss function is log-likelihood (1.1). Decoder-only models are naturally better at this task.

$$\mathscr{L}_{ALM}(x) = \sum_{i=1}^{N} p(x_{i+n} \mid x_i, \ldots, x_{i+n-1}) \tag{1.1}$$

Masked Language Modelling some words in a sequence are masked and the model is trained to predict the masked word using the surrounding context. Sometimes it is refered to as denoising autoencoding. The training objective can be written as follows.

$$\mathscr{L}_{MLM}(x) = \sum_{i=1}^{N} p(\tilde{x} \mid x \setminus \tilde{x}) \tag{1.2}$$

Mixture of Experts (MoE)[50] pretraining enables to pretrain models using less compute. It consists of two components, Sparse MoE layers intead of a dense feed-forward network which each contain a fixed number of ”experts” (which are often FFNs) and a gate network/router determining which tokens are sent to which expert.

1.2.6 Fine-tuning

To perform specific tasks early LLMs required *fine-tuning* (modifying model weights of a pre-trained model to handle task) using labaled data (supervised

fine-tuning). Modern LLMs do not require it but can benefit from it. Fine-tuning can reduce the complexity of prompt engineering and can serve as an alternative to retrieval augmented generation. *Instruction-tuning*[51] is fine-tuning LLMs to provide reponses that are to expectations of humans providing instruction via prompts. Instruction-tuned models usually outperform their base variants. *Self-Instruct*[52] improves instruction-following capabilities of models by using samples generated by itself, which are then filtered, to fine-tune the model.

1.2.7 Alignment

Alignment is the process of steering AI towards human goals, preferences and principles. This does not come naturally through pre-traing and instruction-tuning gets LLMs closer to being aligned. However further alignment is often necessary. *Reinforcement learning from human feedback* and *Reinforcement learning from AI feedback*[53] are two popular approaches. First uses reward model to learn alignment from human feedback and the second one connects an already well aligned model directly to the LLM. Other used approaches are *Direct Preference Optimization*[54] or *Kahneman-Tversky Optimization*[55].

1.2.8 Decoding Samples

A LLM does not generate output tokens directly. It generates a probability distribution. There are multiple strategies how to decode it into a sequence of tokens. *Greedy search* simply takes the most probable token at each step. *Beam Search* takes into account N most likely tokens at each step until the maximum sequence length is reached. Then the most probable seqeuence is chosen. Probabilistic methods include *Top-k sampling* and *Top-p sampling*.

1.3 LLM Usage

After a LLM is trained it can be used to complete various tasks. Some of the task can be solved by direct prompting. This approch is however unable to fully exploit the potential of LLMs and has some shortcomings. This section will describe some of those limitations and several ways how to address.

1.3.1 Limitations

Even though LLMs undergo fine-tuning and alignment to enhance their abilities they are still only predicting the text token. This comes with several limitations when used naively.

State They have no notion of state and don't remember conversations.

Probabilistic They are probabilistic and the same prompt can result in a different output.

Stale information They can access only information used for their training.

Size They tend to be large and resource intensive.

Hallucinations They have no concept of "truth" and were trained on both correct and incorrect data. Sometimes they produce plausible yet factually incorrect answers.

Hallucinations

Hallucinations have recently gained a significant amount of attention. Literature defines hallucinations as "generation of content that is nonsensical or unfaithful to the provided source"[56].

The two basic categories of hallucinations are *intrinsic hallucinations* and *extrinsic hallucinations*. The first describes a direct conflict with the source material while the second describes unverifiable claims. What is considered source varies based on the task. World-knowledge is considered source fo dialogue-based task while input text is considered source for text sumamrization tasks. Sometimes hallucinations can be even beneficial, for exaple in task that require some degree of creativity from the LLM (poem writing).

Methods such as RLHF or instruct tuning have attempted to point LLMs towards more factual outputs however their probabilistic nature and corresponding limitations remain to some extent.

To measure hallucinations in an automated manner two types of metrics are needed. Statistical metrics such as BLEU[57] or ROUGE[58] measuring text simularity or advanced metrics such as PARENT[59] or Knowledge F1[60] are used when there are structured knowledge sources available. Other type of metrics needed are model-based metrics. *IE-Based Metrics* extract information into relational tuples and compare it with sources. *QA-Based Metrics* Assess overlap between generated contend and source using a question answering framework[61]. *NLI-Based Metrics* utilise Natural Language Inference datasets to evaluate truthfulness of a generated hypotheses[62]. *Faithfulness Classification Metrics* create task specific datasets for a more nuanced evaluation.[63].

Human judgement still remains a vital piece despite the advances in automated metrics. One approach is *scoring* when human evaluators rate level of hallucinations on a predefined scale. The second one is *Comparative Analysis* in which evaluators compare generated content against a baseline references.

Other strategies for hallucination mitigation are prompt engineering, model selection and configuration (various parameters can decrease the likelyhood of hallucinations), data management and continuous improvement (keeping track of and analysing hallucinations) and product design (enable user feedback).

1.3.2 Prompt Engineering

Prompt is a textual input provided by user to guide model's output. They generally contain instructions, questions, input data and examples. Prompt engineering a rapidly evolving discipline whose essence is to craft the optimal prompt. It often requires domain knowledge, understanding the model and methodical approach. The process is iterative and exploratory, not entirely dissimilar to traditional machine learning and hyperparameter tuning. Some of the most popular techniques are outlined below.

Chain of Thought (CoT) tries to address LLM's inability to reason. CoT guides the model through the essential reasoning steps. By outlining the steps involved in reasoning the model is steered towards a logical and reasoned response. The two main variants are *Zero-Shot CoT* (instructing the LLM to "think step by step") and *Manual CoT* (providing the model with examples of reasoning)[64].

Tree of Thought (ToT) is inspired by the concept of considering various alternative solutions or thought processes before making a decision. The idea is to utilise "thought trees" in which each branch represents a different line of reasoning. This way the LLM can explore multiple possibilities, much like humans[65].

Self-Consistency relies on a assumption that when the LLM answers in a similar manner to one query mutiple times it is more likely to be an accurate response. It has significant application in fields where factual correctnes is paramount[66].

Reflection involves assessing and potentially revising LLM outputs by itself. After generating the initial response, the model is then prompted to reflect on it while considering factual accuracy, logical consistency and relevant[67].

Expert Prompting involves promting a LLM to assume a role of an expert in a given field and respond accordingly. A key strategy is to prompt a LLM to consider responses from multiple experts, aggregate them and form them into a comprehensive answer[68].

Chains involve linking multiple components in sequence to handle complex tasks. Each component of a chain performs a specific function and the output is fed into the subsequent component. This enables a more complex processing as each component can handle a specific task[69].

Rails refer to guiding the LLM output using rules and templates. This technique is designed to ensure that the reponses adhere to some criteria and enhance relevance, safety or accuracy. Typically rails guide LLM to stick to a topic (Topical Rails), minimize generation of false information (Fact-Checking Rails) or prevent

the LLM from generating content not in line with its guidelines (Jailbreaking Rails).

Automatic Prompt Engineering focuses on automating the process of prompt creation using LLMs in a self-referential manner. A model crates, scores and refines prompts which results in high-quality prompts more likely to elicit the desired response[70].

1.3.3 Retrieval Augmented Generation

Retrieval augemented generation[71] helps to circumvent the innate limitation of LLMs, the lack of up-to-date knowledge. The technique involves extracting a query from the prompt, using it to retrieve relevant information from an external source and adding it back to the prompt to make it available to the model. The model then generates the final response.

1.3.4 Using External Tools

Feeding a model information from an external source is only one potential way to augment it. Generally a LLM can use any external tools. In this context, tool is any external function or service that can be used be an LLM. These can range from simple information retrieval to interaction with databases. The Toolformer paper[72] describes an approach how a LLM can decide what tool to use and when to use it and what parameters to pass to it.

Chapter 2

Agents

2.1 Agents from the Philosophical Perspective

The concept of an agent and agency is as old as philosophy itself and it dates back to the time of Aristotle[73]. Generally, agent is an entity with the ability to act and agency is the manifestation of this property. In a more narrow sense, the actions have to be intentional and it has to posses beliefs and intentions in addition to the ability to act, for an entity to be considered an agent[74]. Agent can be not only a person but also any virtual or inanimate object. Using the general definition AI systems can be considered agents. While in the more narrow sense the question is more complicated with some arguing in favour, some against. With the recent improvement of language models emergence of an artificial intentional agent appears to be more promising. However it still depends on how strict one is with their definition of an agent.

The first usages of AI in agents date back to the 1980's[75]. Ever since then there has been an interest in AI agents. Wooldridge et al.[76] even define AI as a subfield of computer science which aims to construct an agent exhibiting intelligent behaviour. Therefore agent can be considered a central concept within AI. However the term means something slightly different than it does in philosophy. Instead of focusing on whether the agent is literally thinking and possesses a mind of its own the focus is on whether it exhibits properties of such.

2.2 Technology in AI agents in time

Both AI and AI agents have changed over the course of the last few decades and the technology reflects it.

The first AI agents were so called Symbolic Agents utilising at the time predominant approach in AI, symbolic logic[77]. This approach used logical rules

and symbolic representation to enable the agents to think and remember. Typical example of such agent is a knowledge-based expert system. Their disadvantage is their very limited ability to handle uncertainty and large scale real world problems.

A different paradigm were Reactive Agents. They were much simpler in nature than Symbolic Agents focusing on the interaction between the agent and the environment. This enables the agent to respond much faster.[78] These agents usually employ a sense-act loop, quickly sensing and reacting to the enviroment. This design prioritises direct mappings over deliberate reasoning which comes with its advantages and drawbacks. On one hand they require fewer computational resources, on the other they lack the high level decision making abilities of Symbolic Agents.

As computational capabilities and data availability increased so did interest in simulating interaction between agents and their environments. Thus researchers have started utilizing reinforcement learning techniques to train agents to tackle harder and more complex tasks[79]. Initially fundamental techniques such as policy search or value function were used to train agents based on their interactions with the envrionment.[80] As deep learning grew more popular new techniques became available. These deep reinforcement learning techniques which combined deep learning and reinforcement learning enabled agents to learn policies for problems with highly dimensional input. This has led to significant breakthroughs such as AlphaGo[81] or DQN[82]. This approach enables the agent to autonomously learn in uknown environments. There are however disadvantages, the training takes a long time, the sample efficiency is low and it can be unstable when applied to complex real world problems.

To counteract reinforcement learning's training process which requires lot of data and time in addition to not being able to generalise well[83], researchers have introduced transfer learning to speed up the agent's ability to learn new tasks[84]. Thanks to it learning new tasks is faster and generalization capabilities were improved. In addition, meta-learning has been introduced as well[85]. Meta-learning focuses on "learning how to learn" enabling the agent to infer optimal policy faster with a smaller number of samples. Transfer learning however fails when the source and target tasks differ too much and the large amount of pre-training required by meta-learning make it hard to create a universal learning policy.

In the last few years LLMs have grown in popularity thanks to their impressive capabilities[20],[22]. Naturally researchers have started using them to construct AI agents. To be precise they use them as a "brain" of the agent while expanding their action and perception space using other techniques. These agents exhibit reasoning capabilities similar to those of Symbolic agents using prompting techniques such as Chain of Thought[86]. They learn from their environment as

reactive agents do. And their ability to few-shot or zero-shot generalize is also great since they are already trained on a huge amount of data[19].

2.3 What are agents?

In general, LLM-agent is any software that uses LLM's to control the flow of the application[87, 88]. Each agent can be considered to have a "brain", some way to percieve the environment and some way to enact actions. In following paragraphs each of those parts will be explored in detail.

2.3.1 Perception

Both humans and animals rely on all their senses to gather information about the environment. The information is then processed in the brain and a decision is made. Therefore is critical for agents to be able to consume not only text, which is the default form of input for LLMs, as well as other media such as sounds or images to understand the environment better and make better informed decisions. Text is a key mode of inter-human communication and LLMs already have the ability to communicate that way.

Visual Input

Another very rich source of information is visual input. LLMs however lack the ability to work with it. This hinders their ability to understand the whole context in some tasks. This can however be remedied by multiple techniques. One of the possible techniques is to generate a caption for the image and feed it to the agent[89]. This doesn't require any further training. It however loses a lot of information, has low bandwidth and cam introduce biases. Another option is to use transformers[4]. Image is first divided into small fixed-size patches which are (after a projection) fed as an input into a transformer.[90] Calculation of self-attiontion between the tokens then enables to integrate the information across the entire image. Some models have tried to integrate the image encoder and LLM and train them both at once.[91] The abilities of the resulting model were good but the training was very resource intensive. To strike balance between model performance and training complexity a either (or both) the LLM or the pre-trained image encoder can be freezed during training[92]. To enable the LLM to understand the output embeddings from the encoder a learnable translation layer is needed.

Audio Input

Sounds is another vital component of understanding the world. There are multiple models that consider sound a separate modality[93]. These however often excel at a single task. To mitigate that, agents can use LLM as a control hub and invoke existing tooling in order to process audio. For instance AudioGPT leverages multiple models that have achieved great results in audio-related tasks. Another option is to use the spectrogram, a representation of the frequency spectrum of the audio signal over time.[94] For a finite audio sample, this can be viewed as a flat 2D image. Therefore similar approach to handling image data can be used.

2.3.2 Brain

This is the part of the agent where LLM's are the most prominent. The brain facilitates undrestanding of the users desires, using and improving agent's knowledge, recollection and storing of memories, planning and decision making and generalisation to unfamiliar tasks. Each of those aspects will be explored.

Natural Language

Natural language as a medium contains a lot of information. Information that apparent at first sight and also information that is hidden such as beliefs, desires and intentions of the speaker[95]. Thanks to the language understanding and generation capabilities of LLMs agents can profficiently communicate in multiple languages and comprehend the conversation in depth enabling humans to understand them easily and interact with them[22].

One of the foundations of communication is the ability to communicate in turns. And LLMs, which are the core of the agents, excel at it. Multi-turn conversations have two properties. They are interactive, lack continuity and involve multiple speakers. And may include multiple topics and redundant information making the interaction more complex.[96] The conversation can be split into 3 steps. Understanting the history, deciding on the action and generating a response.

LLM's language generation capabilities are exceptional[22], can produce text in multiple languages[97] and can even adapt to the style of the prompt[98]. They perform well with respect to dialogue metrics such as content, relevance and appropriateness. They not only copy the training data but exhibit creativity by generating diverse texts which are sometimes more creative than text written by humans[99]

Even though models understand instructions well enough they are not able to to fully emulate human dialogues and utilise all the information provided

in language[100]. Humans grasp implied meanings easily while agents need to formalize it into a reward function which allows then to choose the correct option[101]. For construction of the reward function multiple methods can be used, one uses feedback[53], some recover it from descriptions and use actions space as a bridge[101]. Others argue that human behaviour can be mapped to choice from a set of implicit options and this formalism can be used with agents[102].

Memory

Memory is where agents stores its past observations, actions and thoughts. These memories are then used in further decision making. They enable the agent to draw from its past experiences and revisit past strategies if needed. As the memory grows over time and is prepended before new input the size starts to become a problem due to the technical limits of the transformer architecture. Another challenge is that having too many memories might hurt the agent performance since finding connections beteween related topics gets harder with the amount of memories.

To solve the first problem, the size of memories, there are multiple possible solutions. One of them is to simple raise the length limit of Transformers. That can be achieved by modifying the attentiom mechanism[103], text truncation[18], input segmentation[104] or emphasizing key parts of the text[105]. Another strategy is memory summarization. This way the agent can effortlessly extract the important piece of information from its memory. To do so, various prompting methods can be used to integrate memories,[106] reflection can condense memory representations [107] or hiearchical methods transform memories into snapshots and summaries[108]. Other option is to compress memories using more suitable datastructures. Multiple methods employ embedding vectors for memories, plans and dialogue histories[108]. Others transform it into triplets of format concept-relationship-concept[109] or even view memories as unique data objects[110]. And finally multiple methods integrate LLMs with SQL databases to store memories in there[111].

How can an agent retrieve the most suitable memory? Some methods focus on automatic retrieval using weighted sums of three metrics, recency, relevance and importance[107]. Other consider memories as interactive objects which can be modified, moved, deleted or combined. The user can then manipulate memories and influence what does the agent ”remember”[110]. Finally, there are methods that allow memory operations using specific commands by users[111].

Knowledge

Agents knowledge is endoced in the parameters of the underlying LLM. The LLMs are trained on massive amounts of data and in theory it is possible for an LLM to comprehend everything there is to comprehend in natural language[112]. Knowledge can be divided into 3 types.

The first type is *linguistic knowledge*[113]. It encompasses morphology, syntax, semantics and pragmatics. It defines which sentences are possible and which are not. It is required to enable the agent to be able to partake in multi-turn conversations.

Other type is common sense or *general knowledge*[114]. This is the type of knowledge humans learn at early age such us ”umbrella protects from rain” or ”don’t touch a hot pan”. This type of information is usually not mentioned explicitly. Agents and models that do not posess this kind of knowledge may fail to understand the task or make incorrect decision.

The last type of knowledge is *domain specific knowledge*, such as programming, mathematics or medicine knowledge. It is necessary for solving problems within a particular domain effectively.

LLMs are great in acquiring, storing and utilising knowledge[115]. There are however some problems. First, the knowledge may become incorrect over time due to the data it’s been trained on becoming obsolete. This can be solved with re-training. However that requires data and a lot of time and computational resources. It can even cause the LLM to catastrophically forget[116]. There are methods to locate and modify a specific piece of knowledge within the LLM but they are still not researched well enough[117]. Another problem is that LLMs can generate content that contradics facts, so called hallucinations[118]. This prevents them from being used in factually rigorous tasks. This can be improved by either providing the LLM model able to use tools[72] to use or providing the developer with some measeure of trustworthiness for the LLM’s response[66].

Reasoning

There are different opinions on the LLMs ability to reason, some argue that they have the abilty during pre-training or fine-tuning[119], others think that they gain it after reaching a certain scale in size[120]. Specificatlly the Chain-of-Thought[64] method brings out the reasoning capabilities of LLMs by guiding them to generate reasonings before outputting the final answer. Other strategies that improve the reasoning capabilities of LLMs include self-consistency[121], self-polish[122], self-refine[123] and selection-interference[124].

Planning

To solve complex challenges an agent needs the capacity to plan[100]. That is enabled by its ability to reason. Agents use it to decompose a task into smaller more digestable subtasks for which they create plans[125]. Later they can even inspect the plans and change them on the go. Typicall planning has two phases. During plan formulation the task is usually decomposed into multiple sub-tasks. This can be done in multiple manners. Either the entire plan is prepared and then executed[126]. Or using Chain-of-Thought[64] prompting makes the agent plan one subtask at a time and adapt on the go. Other methods utilise hiearchical planning or utilise reasoning steps to derive a plan[127]. After a plan is formulated it has to be reflected upon. For that they can either use an existing model[67] or engage with the user to provide feedback and point out problems and misunderstandings[128]. They can also get feedback from the environment[129].

Generalization

Intelligence isn't defined by a specific domain or task. It encompases a broad range of skills and abilities[130]. With large amount of data for pre-training and minimal amount of data for fine-tuning LLMs demonstrace excellent performance on downstream tasks[131]. This task specific fine-tuning lacks the versatinilty and can't be easily generalized to different tasks. However LLM-based agents do not serve as a static knowledge repository. They exhibit learning capabilities enabling them to quickly adapt to unseen tasks[20].

2.3.3 Actions

After the brain is fed the perceptions it processes them and comes to a conclusion. It now needs to act. The most simple response is a simple textual response of which an LLM is perfectly capable.

Tools

As humans do use tools to solve tasks more efficiently, so can agents[132]. They can decide that they need some external tool1 to accomplish a given task. Tools can help the LLM to overcome its innate shortcomings (e.g. not remembering everything from training data[133], hallucinations[134], limited expertise in specific domains[135]). As the "thinking process" of LLMs is not transparent they are unusable in more sensitive fields such as healthcare or finanicial services. They are also vulnerable to adversarial attacks. To contrast this, agents can use pluggable tools to enhance their expertise and make themselves more useful for

domain specific tasks. Tools also make the agent more resilient to sligth input variation and adversarial attacks[132].

To use tools the agent has to learn about how and when to use them. Without it the increase in capabilities would be minuscule. Since LLMs do well in zero-shot and few-shot learning task they can be fed a prompt containing either description of the tool and its parameters (zero-shot) or examples of the tools usage (few-shot) in order to explain them how to use the tool in question[136, 72]. This parallels how humans learn about tools, be reading a manual or observing others as they use them[132]. However a single tool is rarely enough to solve a task therefore an agent should always split tasks into a more digestable subtasks.

Tools are usually designed with humans in mind, not agents, which might hinder their ability to use them. Therefore agent-specific tools are needed, they shoud be modular and have input and output formats more suitable for agents. LLM-based agents can also create tools themselves. When provided with instuctions and demonstrations they can integrate existing tools or generate code for a completely new one[132]. Such tool can be then packaged along with decription and demonstration and be available for other agents to use[137].

Tools enable agents to use various external resources during their reasoning and planning[72]. This provides them with high quality information and expertise. Search based tools can improve the scope and quality of knowledge via various databases, knowledge graphs or website while more domain-specific tools increase agent's expertise in that one particular domain[138]. There are also tools that generate SQL code to query databases, search the web, intepret python code to improve mathematical capabilities or chemical experiments[139, 24, 111, 140]. In multi-agent systems tools for communication (such as emails) may find their use if security constraints require it[132].

Chapter 3

Metaprogramming

This chapter introduces metaprogramming as it encapsulates what MetaGraph does. This chapter is inspired by this survey paper[141].

Metaprogramming refers to writing Metaprograms. These are computer programs that can handle programs as data which enables them to modify them or generate completely new ones. It is not a new concept as it spans back several decades, dating back to early Lisp macros. The source language is called *meta-language*, the target language us called *object language*. When both are the same, it is a case of homogenous metaprogramming, when they differ it is a case of heterogenous metaprogramming. Following section will provide a brief overview of various metaprogramming frameworks and how they can be classified.

3.1 Models

First way how to classify metaprogramming frameworks is by the method they practice metaprogramming in given language. This is called metaprogramming model.

3.1.1 Macro Systems

Macro systems use process called *macro expansion* to map sequences in input file to output-sequences defined by the user. This process is applied iteratively until there are no macro invocations left in the file. Then the macro-free program is sent for translation. Macro Systems can be divided into two distinct categories. *Lexical macros*, which operate on lexical level, meaning they are language agnostic. And *syntactic macros* which can be used only within one language since they are aware of the language's semantics and syntax. They are typically a built-in feature of the language. Other property to classify macro system by are whether or not they

can use algorithmic computations to generate their output (procedural macros) or they use plain pattern matching. Last important property is whether they are *hygienic*, meaning that their expansion doesn't introduce unintended name conflicts. This problem is refered to as variable capture. Typical example of lexical macro system is the C-preprocessor[142]. First language to offer syntax-based macros was LISP[143]. Scheme was the first to provide hygienic macros[144].

3.1.2 Reflection Models

When a computational system can not only reason about itself but also modify its structure and behaviour at runtime, it's called *reflection*. This means that a reflective system can modify its behaviour based on the needs of its execution. The main two concepts of reflection are *introspection* and *intercession*. Introspection enables the system to examine itself while intercession enables it modify itself. Another key distinction is between *structural* and *behavorial* reflection. Structural reflection is abilty of a program to access the representation of its structure (classes, methods, fields,...). Behavorial reflection, on the other hand, can access dynamic representations of its aspects (variable assignments, object creation, etc.). These two distinctions are orthogonal, one relates to the type of access given to the self-representation. The other relates to the type of representation itself. Many popular languages such as Java or C# provide tooling enabling reflection.

3.1.3 Metaobject Protocols

Metaobject Protocols (MOPs), in their original sense, are interfaces to the language enabling transformation of its original behaviour and implementation. MOPs of Smalltalk and CLOS of Lisp are the most prominent examples. In a more relaxed definition MOP allows accessing and manipulating the structure and behaviour of objects within the object system. Classes are considered instances of metaclasses (metaobjects) which are responsible for the overall behaviour of the object system and expose an API to modify it. Many languages such as Python, Groovy or Ruby provide functioanalities based on metaclasses.

3.1.4 Aspect-Oriented Programming

This paradigm extracts cross-cutting concerns into modular units called aspects. These contain information about additional behaviour (advice) that should be added to the base program using that aspect. They also contain information about program locations where the behaviour should be inserted (join points). These are generated based on some matching criteria (pointcuts). Weawing is the process of combining the base program with aspects and may take place

both during compilation and during execution. The first case is so-called static AOP and is usually done using a external compiler (aspect weaver). The other is dynamic AOP in which the bytecode is instrumeted to support weaving aspect code. First to support this model was AspectJ which brought this approach to Java[145]. AspectC++ was the first implementation for C++[146]. Spring is one of the more known representatives of this model[147].

3.1.5 Generative Programming

Generative programming is a "software development paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge". Within metaprogramming, generative programming programs manipulate and generate other programs based on some alorithm and program representation. This closely resembles macro systems, however the key distinction is that in macro systems there is little difference between normal and meta code while in generative programming the difference is clear. There are several sub-categories.

Templates

In C++ metaprogramming is achieved by instantiating template code with specific parameters (types or compile time constants). This produces concrete C++ code which is then compiled.

AST Transformations

Most typicall approach is to provide tooling for AST creation, composition, traversal and transformation to gain metaprogramming capabilities. It is often used in conjunction with quasiquotation. Systems supporting this approach are Groovy,[148] Mython[149] or the Jakarta Toolset[150].

Compile-Time Reflection

Some systems enable compile-time reflection to facilitate generating based on existing code while ensuring static-type safety. Systems supporting this approach are cJ[151], Genoupe C# extension[152] or SafeGen[153].

Class Composition

Other systems focus on composition using mixins[154] or traits[155]. Typical example is Metatrait Java which provides user-customizable traits[156]. These enable compile-time pattern-based reflection.

3.1.6 Multistage Programming

Multistage programming divides program into levels of evaluation and makes tham accessible to the developer using so called staging annotations. These explicitly specify the evaluation order of the program computations and thus enable creation of delayed computations (future stage) and their execution in this stage. This way MSP is closely related to program specialization and partial evaluation since the explicit evaluation order enables specializing generic highly parametrized programs with very little runtime overhead. It can be also viewed as a special form of program generation. This way it's related to procedural macro systems with the difference that macro systems use extra syntax for definitions and normal for invocations while MSP uses extra syntax at the callsite to execute delayed computation. It was frist applied in MetaML[157] and followed by MetaOCaml[158].

3.2 Phase of Evaluation

Antother criteria based on which metaprograms can be categorised is when they are evaluated. Metaprograms can belong into multiple categories but they mostly belong to one or two.

3.2.1 Preprocessing-Time Evaluation

First option is to apply metaprogramming is while pre-processing the original program before translation. This is called preprocessing-time metaprogramming. Since the metaprograms present in the original source are evaluated during preprocessing the resulting code contains only normal program code. Which means that it is not necessary to extend the compiler or interpreter. Systems that operate this way are called source-to-source preprocessoers. Sometimes, when the compiler is available as a library, the compiler can be bundled with the pre-processor to output binary code directly. This blurs the line between preprocessing-time and compile-time evaluation. The distinction is mostly practical and based on desgin and implementation issues.

3.2.2 Compilation-time Evaluation

Another option is to apply metaprogramming during compilation. This requires extending the compiler to support metacode translation and execution. This is usually done via compiler plugins, syntactic additions, procedural or rewrite-based AST transformations, or multistage translations. Moreover metacode execution can be done using an interpreter withing the compiler or running a binary that is generated by a compiler.

3.2.3 Execution-time Evaluation

Metaprograms can be also applied during the program execution itself. This is called *runtime metaprogramming*. For a system to support it, it needs to extend the language execution system and offer runtime libraries enabling runtime code generation and execution. It is also the only case where metaprogramming can extend the system based on runtime state. In interpreted languages this concept is closely coupled with the existence of `eval` function. In interpreted languages with macros, macro expansion and execution of the generated code are executed as one operation.

3.3 Metaprogram Source Location

Another way how to differentiate various systems is based on where does the metaprogram code reside. It can either be a separate file or it can be embedded in the original source code.

3.3.1 Embedded in the Subject Program

Typically, metaprograms are part of the program they transform, blended with normal code. Macros or generative programming templates are placed placed together with the normal code. MSL uses annotations to distinguish its directives from normal code. Definition of reflection or MOP based transformation is also done within the language. And finally, static AOP metalanguages typicall extend the host language so that to code can be intersperesed. There are three transformation options available.

Context Unaware These do not need to know the code genration context, only input parameters are enough and the ouput directly replaces the invocation.

Context Aware In this case the transformation logic takes the metaprogram location context. Typically it means providing the AST node as an extra

input which can be read or modified by the metaprogram. This enables the metaprogram to transform code at multiple different locations reachable from the initial context.

Global This approach enables the metaprogram to change the behaviour of the entire program. It is typically encountered in systems with reflection, MOPs and AOPs.

3.3.2 External to the Subject Program

Metaprograms can alse reside outside of their subject programs. They are specified as separate transformation programs and then supplied to the compiler as extra parameter when compiling the target program.

Chapter 4

Frameworks

This chapter provides a brief overview of tools and frameworks for creation of LLM-based. One of them, LangChain, is then described in greated detail because MetaGraph extends it.

As LLM-based agents are software as any other there is a myriad of frameworks and tools to make the job easier. These can be split into two distinct groups. There are various low-code or no-code solutions that enable people with lesser technical skills. They enable the user to create a functioning agent without writing a single line of code, usually by providing a drag and drop UI or a simple configuration. The other group are typical frameworks and libraries requiring the developer to write code in order to build the agent. In the rest of this chapter some frameworks will be very briefly introduced and then a more in-depth description of LangChain platform will be provided as this thesis builds upon it.

Frameworks for creating LLM-based agents are created both by large corporations and small startups. One of the large companies that has its own framework for agent is Amazon with its **Amazon Bedrock**[1] platform. This platform enables the user to create agents via a simple GUI. The user selects a LLM to use, creates a prompt and connects it to an action group (API, function or AWS Lambda function). It also supports multi-agent collaboration.

Another large company that has a framework for LLM-agents available is Open AI with their **OpenAI Agents SDK**[2], a lightweight framework available both in Python and JavaScript. The framework revolves around 5 basic concepts. Agents (instruction, tools, guardrails and handoffs), handoffs (tool call to transfer control between agents), guardrails (safety checks), sessions (conversation history management) and tracing (tracking of agent runs).

CrewAI[3] is a framework made by company of the same name. It provides

[1] `https://aws.amazon.com/bedrock/`

[2] `https://openai.github.io/openai-agents-python/`

[3] `https://www.crewai.com/`

a library (and in the enteprise version a Web GUI) to create crews of agents. Crew organizes the overall operation, agents work on specialized tasks, process orchestrate collaboration and tasks represent individual assignments.

AutoGen[159] and **AutoGen Studio**[160] are a framework and no-code platform by Microsoft for creating agents. It provides the Autogen AgentChat library which enables building conversational single and multi-agent applications. Its built around AutoGen Core library which is an event-driven framework for building multi-agent AI systems and can be also used to do reseach on multi-collaboration.

Rivet[4] is a GUI tool for creating agents. It's built around the idea that each agentic workflow can be viewed as a state graph and the user is creating nodes containing application logic and edges to direct control flow in a GUI.

4.1 LangChain Platform

LangChain is a very popular (111k stars on GitHub[5] as of writing this thesis) ecosystem of components created by company of identical name and is used by many companies ranging from huge corporations such as Google to small startups[161]. It aims to aid all phases of LLM application development. Its main selling points are following[162]

Standardised interface for components with the number of models and related components each having a different API it simplifies development of AI applications by providing a standardised interface so that switching of e.g. underlying LLM requires very few code changes

Orchestration with applications growing more complex, incorporating multiple models and non-linear control flow, there is a need for a tool that will connect various parts of the application in an efficient manner

Observability and evaluation with growing complexity of application, understanding what is happening inside is becoming harder and. It is however needed to steer the development in the correct direction.

Each of those selling points is covered by one main component of LangChain. First by the original LangChain library, the second by the LangGraph library, and the last by the LangSmith platform.

[4] `https://rivet.ironcladapp.com/`

[5] `https://github.com/langchain-ai/langchain`

4.1.1 LangChain

LangChain itself provides interfaces for components necessary to build AI applications, most notably chat models and tools. The library exists in Python and TypeScript versions.

The core building block of LangChain is the `Runnable` interface. It is implemented by many types of components, language models, output parsers, retrievers, tools or LangGraph graphs. It provides the component with the ability to be invoked (called with one argument), batched (efficiently transform multiple argument), streamed (efficiently transform multiple arguments while enumerating results one by one), inspected (see input and output types) and composed into more complex pipelines using the LangChain Expression Language (LCEL). All methods of execution support both sync and async interface. They also accept an optional parameter `RunnableConfig` which can contain various metadata related to the execution, callbacks, identifier or runtime config for the runnables. This parameter then gets automatically propagated to all the runnables that are executed as part of the original runnable.

The key component to LLM applications is the chat model. Along with the `Runnable` interface, all chat models implement the `BaseChatModel` interface. This provides the models with two important features for modern LLM applications. The ability to provide tools for the underlying model to use, if it does support it.

```
# Tool creation
tools = [my_tool]
# Tool binding
model_with_tools = model.bind_tools(tools)
# Tool calling
response = model_with_tools.invoke(user_input)
```

And the ability to return a structured output instead of an unstructured text.

```
# Define schema
schema = {"foo": "bar"}
# Bind schema to model
model_with_structure = model.with_structured_output(schema)
# Invoke the model to produce structured output that matches
 ↪  the schema
structured_output = model_with_structure.invoke(user_input)
```

The models use messages as inputs, these can have multiple types: system, user, assistant or tool. The result is also a message, or an object with defined schema if using structured output.

Another part of a modern LLM application are tools. Tools encapsulate function in such way thay it can be passed to a chat model to aid it in solving tasks. In LangChain, each tool is a python function with a schema containing its description, expected arguments and expected outputs. To simplify its creation a `@tool` decorator is provided which infers all of these based on the name, documentation comments and typehints.

```
@tool
def multiply(a: int, b: int) -> int:
    """Multiply two numbers."""
    return a * b
```

After a tool is bound to a model using the `bind_tool` function the model can now use it to solve its task. However, it does not do it directly. Instead, it returns a message where the property is set and contains a list of tools with their associated arguments. The user then executes the tools with provided arguments and it returns a tool message. This then gets passed as another input to the model and now it can use the knowledge gained this way.

4.1.2 LangGraph

LangGraph is a low level orchestration framework for building LLM agents. It provides abstractions to build any long running stateful workflow or agent. It does not abstract working with models or prompts. The benefits it provides are following.

durable execution Agents can faile at any point for any imaginable reason and need to resume from their last successful step

human in the loop Easy integration of human, be it for providing feedback, approving tool calls or manually editing state.

memory Support for both short and long term memory.

debugging Native integration with LangSmith allowing for visualisation and tracing.

LangGraph offers two distinct APIs, Graph API and Functional API. The former is built around Nodes, Edges and State and requires your application to be built with LangGraph in mind. The latter provides a more lightweigth integration of LangGraph into an already existing application using `@entrypoint` and `@task` decorators. It however does not offer all the benefits of the Graph API.

Runtime

The runtime is the same regardless of the used API. It is powered by Pregel. Pregel is an execution runtime built around *actors* and *channels* based on Google's algorithm of the same name[163]. Actors read data from channels, transform it and then write to channels. Channels provide communication between actors. In each step actors which's input channels have been upated are executed and update their output channels. This repeats until there are no actors with updated input channels.

Graph API

The Graph API is built around 3 key components.

State shared structure representing the state of the application

Node container for logic of the application

Edge connects Nodes to mark the order of execution

Building a graph starts with a `StateGraph` object which requires a State schema to be initialised. It can be either `TypedDict` or `BaseModel` from Pydantic library.

The State schema describes the state of the application and there can be multiple schemas defined, input, output, overall and then each node can have its own input schema. After all nodes and edges are added, the graph can be compiled.

```
builder = StateGraph(OverallState, input_schema=InputState,
 ↪  output_schema=OutputState)
...
graph = builder.compile()
```

Another important concept are reducers. These are functions that are used to update the state when a new value is returned from a node. By default the value is simply overwritten. However if specified by attaching it to the original type using the `Annotated` Python type, then the attached function is used to update the state.

```
class State(TypedDict):
    foo: int
    bar: Annotated[list[str], add]
```

Nodes are added to the graph using the `add_node` method and each Node has its name and action. The action should be an aforementioned runnable or a Python function. This is where all aplication logic resides. It should return a state update.

```
def my_node(state: State, config: RunnableConfig):
    print("In node: ", config["configurable"]["user_id"])
    return {"results": f"Hello, {state['input']}!"}

builder.add_node("my_node", my_node)
```

There are two different approaches to connecting nodes together to form a workflow. First is to create edges explicitly.

```
graph.add_edge("node_a", "node_b")
graph.add_conditional_edges("node_a", routing_function)
```

This either adds a simple edge between two nodes or a conditional edge. The second paramter for creation of conditional edges is a function accepting state and returning the name of the node to execute next.

The second way is to create implicit edges. This is done by returning a `Command` from a Node which enables to connect control flow and state modification. In the Command, a following Node can be specified by using the `goto` parameter.

```
def my_node(state: State):
    return Command(
        # state update
        update={"foo": "bar"},
        # control flow
        goto="my_other_node"
    )
```

Another important concept is `Send`. By returning a collection of `Send` objects from a node it enables to execute multiple nodes at once *with different states*. This enables the usage of map-reduce pattern by specifying a concatenating reducer on a particular part of state.

```
class OverallState(TypedDict):
    ...
    jokes: Annotated[list[str], add]

def continue_to_jokes(state: OverallState):
```

```
    return [Send("generate_joke", {"subject": s}) for s in
     ↪  state['subjects']]

def generate_joke(state: OverallState):
      joke = make_a_joke(state['subject'])
      return {"jokes": [joke]}
```

To build multi agent systems one can use sub-graphs. This means using a graph as a node in a different graph. However the question is, how do the graphs communicate, how do they pass the state between each other? One option is to share some (or all) state keys.

```
subgraph_builder = StateGraph(State)
...
subgraph = subgraph_builder.compile()

builder = StateGraph(State)
builder.add_node("subgraph_node", subgraph)
...
graph = builder.compile()
```

The other way is to call the sub-graph directly within a node and provide it with state. This makes their states completely independent of each other.

```
subgraph_builder = StateGraph(State)
...
subgraph = subgraph_builder.compile()

def f(state: State):
      response = subgraph.invoke(...)
      return {"foo": response["bar"]}

builder = StateGraph(State)
builder.add_node("f", f)
...
graph = builder.compile()
```

The Graph API comes with built-in visualisation using mermaid diagrams[164]. This can help with development since visualising the overall flow just from definitions of nodes and edges can be difficult.

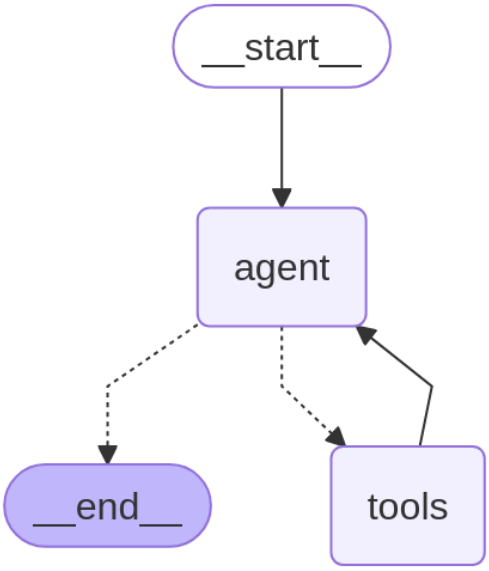

Figure 4.1 Exampe visualisation of a LangGraph agent

Functional API

Functional API makes adding LangGraph features (persistence, memory, human-in-the-loop) to an already existing codebase easier. It does so by providing two key bulding blocks. `@entrypoint` decorator marking function as starting point of a workflow. And `@task` decorator ot mark discrete units of work within a workflow.

```
@task()
def slow_computation(input_value):
    ...
    return result

@entrypoint(checkpointer=checkpointer)
def my_workflow(some_input: dict) -> int:
    # some logic that may involve long-running tasks like API calls,
    # and may be interrupted for human-in-the-loop.
    result = await slow_computation(some_input)
    ...
    return result
```

This way LangGraph features such as checkpointing, human-in-the-loop and others that will be described in the next section. However not all, due to how checkpointing in Functional API works, ”time-travel” is not possible.

The main advantage this approach has over the Graph API is that there is no need to think about state or graphs to define a workflow. Normal Python constructs are enough. However Graph API still is the more feature complete API since it supports (as mentioned above) ”time-travel” and visualisation, which Functional API does not support.

Persistence

Persistence is implemented through checkpointing. When a graph is compiled the checkpointer saves a snapshot of its state at every step of execution. Each checkpoint belongs to a thread and the thread can be accessed even after the execution is done. This enables several capabilities.

First, it enables human-in-the-loop workflows. To relinquish control from an agent to a human there is the `interrupt` function. Its invocation within a node throws an exception and pauses the execution. The user then can review the graph state, potentially modify it, and using a Command with `resume` paramter set re-invoke the graph. The graph will then continue executing in the node where it was before the interruption.

Another capabality is memory. Short-term memory tracks the current converstaion and maintains message history. State also get’s persisted after each Node is executed. Long-term memory enables agents to retain information for longer, across multiple conversations. And unlike short-term memory, it is not thread-scoped. Its organised into custom namespaces to separate memories.

The most interesting sounding capability is ”time travel”. This simply means that even after the agent is finished the user can not only inspect the history. They can also modify the state at any point and fork the execution at an arbitrary point to explore different trajectories.

Lastly it enables fault-tolerance. When execution fails at some node, it can be easily restarted from the last successfuly executed node.

Chapter 5

MetaGraph

In this chapter a metaprogramming system called MetaGraph for easier creation of agentic-workflows is introduced and explained. First a brief motivation will be laid out. Then a general description of the system will be provided along with challenges of implementing it in Python. Finally it will be demonstrated on an example and its limitations discussed.

5.1 Motivation

As outlined in the introductory chapter, creating agentic-workflows represented as graphs can be cumbersome. Moreover it requires the user to specify the control flow in a really unintuitive manner. The flow of data and control flow are mixed together and the control flow is spread in multiple places, the graph definition and within node definitions. Also, defining the state beforehand is unwieldy and forces the developer to think slightly differently then they do when programming normally. The idea is to create is to create a language that will enable developers to create agent-based workflow without the need to think about the graph explicitly but still get the benefits of the underlying runtime (in this case LangGraph). The goal is to get from 5.1 to 5.2 while keeping the behaviour essentially identical.

5.2 Design

The core idea is, that one of the best ways how to describe control flow is the thing that developers already know well. Source code of a procedural programming language. This can be then transformed into a graph by considering each statement a different node and connecting subsequent nodes with edges with some special cases such as conditionals and while-cycles. State also doesn't have

```
section_builder = StateGraph(SectionState,
 ↪  output=SectionOutputState)
section_builder.add_node("generate_queries", generate_queries)
section_builder.add_node("search_web", search_web)
section_builder.add_node("write_section", write_section)

# Add edges
section_builder.add_edge(START, "generate_queries")
section_builder.add_edge("generate_queries", "search_web")
section_builder.add_edge("search_web", "write_section")


# Add nodes
builder = StateGraph(ReportState, input=ReportStateInput,
 ↪  output=ReportStateOutput, config_schema=Configuration)
builder.add_node("generate_report_plan", generate_report_plan)
builder.add_node("human_feedback", human_feedback)
builder.add_node("build_section_with_web_research",
 ↪  section_builder.compile())
builder.add_node("gather_completed_sections",
 ↪  gather_completed_sections)
builder.add_node("write_final_sections", write_final_sections)
builder.add_node("compile_final_report", compile_final_report)

# Add edges
builder.add_edge(START, "generate_report_plan")
builder.add_edge("generate_report_plan", "human_feedback")
builder.add_edge("build_section_with_web_research",
 ↪  "gather_completed_sections")
builder.add_conditional_edges("gather_completed_sections",
 ↪  initiate_final_section_writing, ["write_final_sections"])
builder.add_edge("write_final_sections",
 ↪  "compile_final_report")
builder.add_edge("compile_final_report", END)

graph = builder.compile()
```

Listing 5.1 Open Deep Research in LangGraph

```python
@agent_func
async def section_agent(topic: str, section: Section) ->
 ↪  Section:
    iterations = 0
    improvement_needed = True
    while improvement_needed:
        iterations = inc_iterations(iterations)
        queries = generate_queries(topic, section)
        source_str = await search_web(queries)
        section = await write_section(section, topic,
         ↪  source_str)
        feedback = eval_section(section, topic)
        queries = update_search_queries(feedback, queries)
        improvement_needed = needs_improvement(iterations,
         ↪  feedback)
    return section


@agent_func
async def report_generator(topic):
    report_structure_not_ok = True
    feedback = None
    while report_structure_not_ok:
        plan = await generate_report_plan(topic, feedback)
        feedback, report_structure_not_ok =
         ↪  human_feedback(plan,  feedback)
    sections_with_research, sections_without_research =
     ↪  split_sections(plan)
    written_sections_with_research = [await
     ↪  section_agent(topic,s) for s in
     ↪  sections_with_research]
    formatted_sections_with_research =
     ↪  gather_completed_sections(written_sections_with_research)
    written_sections_without_research = [await
     ↪  write_final_section(s, topic,
     ↪  formatted_sections_with_research) for s in
     ↪  sections_without_research]
    report =
     ↪  compile_final_report(written_sections_with_research,
     ↪  written_sections_without_research, plan)
    return report
```


Listing 5.2 Open Deep Research in MetaGraph

to be defined explicitly, it can be generated dynamically by collecting variables present within the code.

In its most abstract form, the proposed language supports 5 different types of statements:

assignment It should evaluate the expression on the right side, update the state variables on the left side of the statement and move on to the next node in the graph.

if-statement It should evaluate the condition and then direct the execution flow to the appropriate next node based on the condition.

while-statement It should evaluate the condition and then direct the execution flow to either execute the loop or continue further.

map-reduce statement Takes a sequence as a input, each item is then used as a input to a function and then concatenates the results.

return statement Should evaluate the value and put in a special state variable which is used as output from the agentic workflow and end the execution.

The inclusion of the map-reduce construct may seem strange at first. It however is a very useful paradigm for processing collections of data and allows for paralellisation.

To generate a graph from a list of statements, the algorithm iterates through statements one by one connecting it to its neighbours. This can be done in two different manner. The more intuitive way is to process statements in order and connect them nodes, that represent statements that precede them. This approach however makes processing of if and while statements more problematic. Because to point the flow in the right direction based on the condition the nodes would already have to exist. The original version of the algorithm used this approach and it circumnavigated this issue by creating auxiliary nodes. This however made the resulting graph harder to understand due to the number of auxiliary nodes.

The other approach iterates through the statements in a bottom-up manner. This way the node being created is simply connected to the nodes that follow it. The algorithm is recursive in nature and will be described below.

The core loop of the algorithm, as already mentioned, iterates through the statements in the current block, inspects which type of statement it is, processes it with the appropriate function and updates reference to the *next* node.

```
function ProcessStatements(Statements, next = END)
    for statement in Reversed(Statements) do
        if statement is assignment then
            next = ProcessAssignment(statement, next)
        else if statement is map-reduce then
            next = ProcessMapReduce(statement, next)
        else if statement is if-statement then
            next = ProcessIf(statement, next)
        else if statement is while-statement then
            next = ProcessWhile(statement, next)
        else if statement is return then
            next = ProcessReturn(statement, next)
        else
            end with error
    return next
```

Figure 5.1 Core algorithm

Processing of assignments is simple, a new node is created and is connected to the next node.

```
function ProcessAssignment(statement, next)
    node = AddNode(statement)
    AddEdge(node, next)
    return node
```

Processing of assignments is trivial, a new node is created and is connected to the *END* node.

```
function ProcessReturn(statement, next)
    node = AddNode(statement)
    AddEdge(node, END)
    return node
```

To transform if-statement into graph the two branches (lists of statements) need to be processed first. Then a node for the if-statement is created. And finally two conditional edges are created.

```
function ProcessIf(statement, next)
    trueBranch = ProcessStatements(statement.trueBranch, next)
    falseBranch = ProcessStatements(statement.falseBranch, next)
    node = AddNode(statement)
    AddConditionalEdge(node, trueBranch)
    AddConditionalEdge(node, falseBranch)
    return node
```

Processing of while-statement is very similar to if-statement with the difference that the node, that is created for the while statement is considered as the *next* node while processing the body of the while-expression. This way the execution will loop back once the whole body is executed.

```
function ProcessWhile(statement, next)
    node = AddNode(statement)
    body = ProcessStatements(statement.body, node)
    AddConditionalEdge(node, body)
    AddConditionalEdge(node, next)
    return node
```

Processing of map-reduce statement results in multiple nodes. One splits the input sequence into elements, second one transforms the input and the last one collects the results.

```
function ProcessMapReduce(statement, next)
    split = AddNode(statement.values)
    map = AddNode(statement.transformation)
    reduce = AddNode(statement.aggregation)
    AddEdge(split, map)
    AddEdge(map, reduce)
    AddEdge(reduce, next)
    return split
```

5.3 Analysis

This section will provide analysis of considered approaches and explain why a particular approach was chosen.

The first considered approach was to design a language completely from scratch and implement a parser and interpreter for. The clear advantage of this approach is the complete freedom in designing the language which could be

tailored precisely for the usecase. This however does not outweigh disadvantages of this approach. The biggest disadvantage would be that instead of graph theory the developer would have to learn a custom syntax. Moreover there would be no tooling for that language. Therefore it was decided not to pursue this approach.

Another possible approach would be to create tooling within Python that would make creating of graphs easier. This improves upon the previous approach by not forcing the developer to learn a completely new language and make existing available. However it would still require the user to learn how to use a custom API.

The approach that ended up being used was to use normal Python and interpret it slightly differently. Or, to be precise, its subset. The idea is to take a Python function and transform it into a LangGraph graph using an algorithm that is derived from the algorithm introduced in the previous section. Advantages of this approach are that the developer doesn't have to learn any new syntax and all the tooling available for Python (intellisense, type-checking, etc.) should work without any issue. In a way it is a different implementation of the Functional API 4.1.2 but in this case the graph is explicitly constructed and all features that are available to Graph API but not to Functional API are present.

This approach is, from the point of view of metaprogramming classification, considered a generative programming system with AST rewriting.

This is possible thanks to Python being an interpreted language and Python's built in tooling for inspection of runtime objects and parsing of Python's source code. When presented with an instance of `Callable[…]` its source code can be retrieved by using the `getsource` method from the `inspect` module. The retrieved source code can be then parsed into an AST using the `parse` method from the `ast` module.

```
def get_ast(f: Callable[...]):
    source_code = inspect.getsource(f)
    return ast.parse(source_code)
```

5.4 Implementation

This section will decribe the actual implementation and what challenges had to be tackled to make it work.

The created language should support creating graphs using the algorithm described above. Moreover, as LangGraph is used as the backend, there are several additional features that are either needed or make creating agents easier and the implementation should support them as well. First, there is the ability to create

sub-graphs. Next there is automatic config injection since passing a `config` parameter into every function call is tedious. And lastly, before applying the algorithm, the type of State will have to be determined. There are also minor quality of life features such as support for `await` or support for tuple deconstruction and thus the possibility to return multiple values from one function.

5.4.1 Grammar

The grammar of the language is, as is evident from the definition (5.3), a limited version of Python's grammar.

⟨*statement*⟩ ::= ⟨*assignment*⟩ | ⟨*if-statement*⟩ | ⟨*while-cycle*⟩ | ⟨*return-statement*⟩

⟨*return-statement*⟩ ::= '`return`' ⟨*vars*⟩

⟨*assignment*⟩ ::= ⟨*vars*⟩ '`=`' ⟨*list-comp*⟩ | ⟨*func-call*⟩ | ⟨*constant*⟩

⟨*if-statement*⟩ ::= '`if (`' ⟨*expr*⟩ '`) :`' ⟨*statements*⟩ ⟨*elif*⟩?
| '`if (`'⟨*expr*⟩ '`) :`' ⟨*statements*⟩ ⟨*else-statement*⟩?

⟨*elif-statement*⟩ ::= '`elif (`' ⟨*expr*⟩ '`) :`' ⟨*statements*⟩ ⟨*elif*⟩?
| '`elif (`' ⟨*expr*⟩ '`) :`' ⟨*statements*⟩ ⟨*else-statement*⟩?

⟨*else-statement*⟩ ::= '`else:`' ⟨*statements*⟩

⟨*while-statement*⟩ ::= '`while (`' ⟨*expr*⟩ '`) :`' ⟨*statements*⟩

⟨*statements*⟩ ::= NEWLINE INDENT ⟨*statement*⟩+ DEDENT

⟨*list-comp*⟩ ::= '`[`' ⟨*func-call*⟩ '`for`' ⟨*identifier*⟩ '`in`' ⟨*identifier*⟩ '`]`'

⟨*func-call*⟩ :: '`await`'? ⟨*identifier*⟩ '`(`'⟨*vars*⟩ '`)`'

⟨*vars*⟩ ::= ⟨*identifier*⟩ | ⟨*vars*⟩ '`,`' ⟨*identifier*⟩

⟨*constant*⟩ ::= '`True`' | '`False`' | '`None`' | ⟨*number*⟩ | ⟨*string*⟩

⟨*expr*⟩ ::= arbitrary Python expression

⟨*identifier*⟩ ::= any valid Python identifier name

Listing 5.3 MetaGraph language grammar

5.4.2 Implementation Details

As was mentioned in a preceding section, the implementation is based on the proposed algorithm. There were however some challenges that had to be solved first as well as design decision to be done. This section will try to describe some of those challenges and explain conceptually how they were solved.

First, the type of State has to be determined. Given a function, its state consists of its variable and its parameters. The function parameters can be collected using the built-in `inspect.signature(...)` function. Getting the other variable is not as trivial. Fortunately Python provides AST visitor. It can be used to collect all assignemnt statements from the graph and then these assignments can be processed one by one. To do so, the `visit_Assign` method must be overloaded.

```
class CollectAssignmentsVisitor(ast.NodeVisitor):
    assignments: list[ast.Assign] = []

    def __init__(self):
        super().__init__()

    def visit_Assign(self, node):
        self.assignments.append(node)
        return None
```

The first design decision is to use implicit edges for routing, meaning that every node will return a `Command` pointing to the next node. This is done so that it is consitent across all created nodes. Each node can be then annotated during its creation with the `destinations` doesn't have a runtime effect. It only helps when generating a picture of the graph. The only effect this has is that when generating a picture of the result graph all edges are dashed lines instead of only the conditional ones. This is however an inconcsequential detail.

As mentioned in the section about LangGraph, definition of a node function accepts state and config and returns a state update. How to transform an assignment `y = foo(x)` into that form?

First a callable representing `foo` is needed. It can be fetched from the source module of the function which is being transformed into graph (let's call it `f`) using the `getattr` built-in function. The module where `f` is defined can be gotten by using the `inspect.getmodule(...)` function. This whole approach is also the reason why MetaGraph *supports only functions that are defined on the top level in a module*. Now the other task is to correctly map variables from the state (and potentially config) onto the function parameters. done by inspecting `foo`'s signature (`inspect.signature(foo)`). All put together the logic (simplified) looks like this.

```
def transform(f, variable_name, next_node):
    callable_f = get_callable(f)
    signature = inspect.signature(callable_f)
    def wrapper(state, config):
        result = callable_f(**create_params_dict(signature,
         ↪ state, config))
        return Command({variable_name: result}, goto:
         ↪ next_node)
    return wrapper
```

Another task that needs to be solved is how to get from a pythonic list comprehension to the map-reduce pattern supported by LangGraph. That can be modeled as two separate nodes in the graph. The first one takes the input sequence and uses `Command([Send(...) for x in params])` to split the computation. Then the following node does the acual computation and wraps the result in a list. The lists are then concatenated as the state variable that is used for the result has a special reducer attached to it which supports both concatenation (for list comprehension) and overwriting (for other usages).

Another challenge is sub-graphs. Creating the sub-graph is easy, the algorithm can be applied to the appropriate callable and its output (a LangGraph's `CompiledStateGraph`) can be then invoked. The tricky part is knowing when to transform a callable into a simple node and when make it into a completely new graph. The solution is simple thanks to Python being able to attach an arbitrary property to any object at runtime. It is enough to attach a property designating that a function is supposed to be made into a graph. And that is easily done using a decorator. Then, when inspecting the callable during creation of a node, one can easily decide whether to create a standard node or to create a subgraph.

5.5 Showcase

In the section MetaGraph will be showcased on 2 examples. A simple one, Retrieval Augmented Generation, and a more complex one, Open Deep Research. One could

Both of these examples were prepared by LangGraph as a traditional graph based workflow and then transformed to be compatible with MetaGraph. For that, several things are necessary. Functions must be modified to accept separate fields from state instead of the whole state and return plain values (or tuples), intead of state updates. And the whole logic must be reworked to be more in line with procedural programming.

5.5.1 Retrieval Augmented Generation

The particular version of RAG that is used in this example is Adaptive RAG[165] and is taken from LangGraph tutorial repository[1]. This approach enables to change approach to RAG based on what is the query like. This particular example routes between web-search for information about recent events and self-corrective RAG for queries about data in the index (3 blogposts related to LLMs).

As can be clearly seen in the code excerpt (5.4) there are 5 nodes and the edges tend to loop back. That is a type of flow that is not suited for procedural programming. Nevertheless it was successfuly rewritten (5.5) into three functions and one while-loop. To model the non-linear nature of the original flow a `destination` variable had to be introduced. This varible controls which step will be done next and is used in a way that is similar to go-to. This approach can be used to model other highly non-linear workflows.

When looking at the generated graphs decribing the workflow (5.2 and 5.3) the original one is clearly simpler. That is mostly thanks to it being able to re-use nodes. The MetaGraph solution could limit the number of nodes created. That would however come at the cost of code legibility.

5.5.2 Open Deep Research

The other example is Open Deep Research. Open Deep Research is an open-source implementation of OpenAI's Deep Research [166]. The used open-source implementation is available on LangChain's Github [2].

This agent independently performs research based on a prompt on behalf of the user. Hundreds of online sources are searched and analyzed and a comprehensive report is generated.

When inspecting the original source code (5.1) it is hard to follow the flow of logic as it consists of nodes and edges only. Following the the flow in the MetaGraph implementation (5.2) is much simpler as it ordinary Python code. Since this workflow much more linear than the previous example it does not require any special variable as go-to.

The generated graph for the MetaGraph version (5.4) is clearly simpler than the one present in the previous example (5.3).

[1] `https://github.com/langchain-ai/langgraph/tree/main/docs/docs/tutorials/rag`

[2] `https://github.com/langchain-ai/open_deep_research/tree/main/src/legacy`

```
workflow = StateGraph(GraphState)
# Define the nodes
workflow.add_node("web_search", web_search)  # web search
workflow.add_node("retrieve", retrieve)  # retrieve
workflow.add_node("grade_documents", grade_documents)  # grade
 ↪  documents
workflow.add_node("generate", generate)  # generate
workflow.add_node("transform_query", transform_query)  #
 ↪  transform_query
# Build graph
workflow.add_conditional_edges(
    START,
    route_question,
    {
        "web_search": "web_search",
        "vectorstore": "retrieve",
    },
)
workflow.add_edge("web_search", "generate")
workflow.add_edge("retrieve", "grade_documents")
workflow.add_conditional_edges(
    "grade_documents",
    decide_to_generate,
    {
        "transform_query": "transform_query",
        "generate": "generate",
    },
)
workflow.add_edge("transform_query", "retrieve")
workflow.add_conditional_edges(
    "generate",
    grade_generation_v_documents_and_question,
    {
        "not supported": "generate",
        "useful": END,
        "not useful": "transform_query",
    },
)
```

Listing 5.4 LangGraph RAG implementation

```
@agent_func
def retrieve_and_grade(question):
        documents = retrieve(question)
        documents = grade_documents(question, documents)
        destination = decide_to_generate(question, documents)
        return destination, documents

@agent_func
def generate_and_grade(question, documents):
    generation = generate(question, documents)
    destination =
     ↪  grade_generation_v_documents_and_question(question,
     ↪  documents, generation)
    return destination, generation


@agent_func
def rag(question):
    question_classification = route_question(question)
    if question_classification == "web_search":
        documents = web_search(question)
        destination, generation = generate_and_grade(question,
         ↪  documents)
    else:
        destination, documents = retrieve_and_grade(question)


    while destination != "end":
        if destination == "generate":
            destination, generation =
             ↪  generate_and_grade(question, documents)
        else:
            question = transform_query(question)
            destination, documents =
             ↪  retrieve_and_grade(question)

    return generation
```

Listing 5.5 MetaGraph RAG implementation

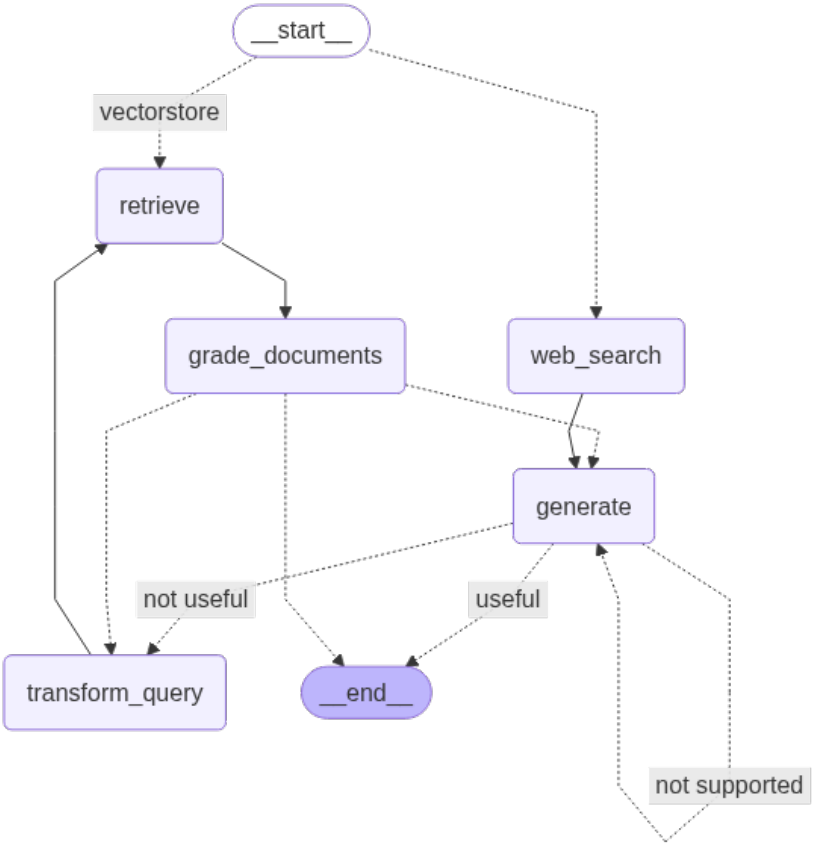

Figure 5.2 Graph generated for the LangGraph approach to RAG

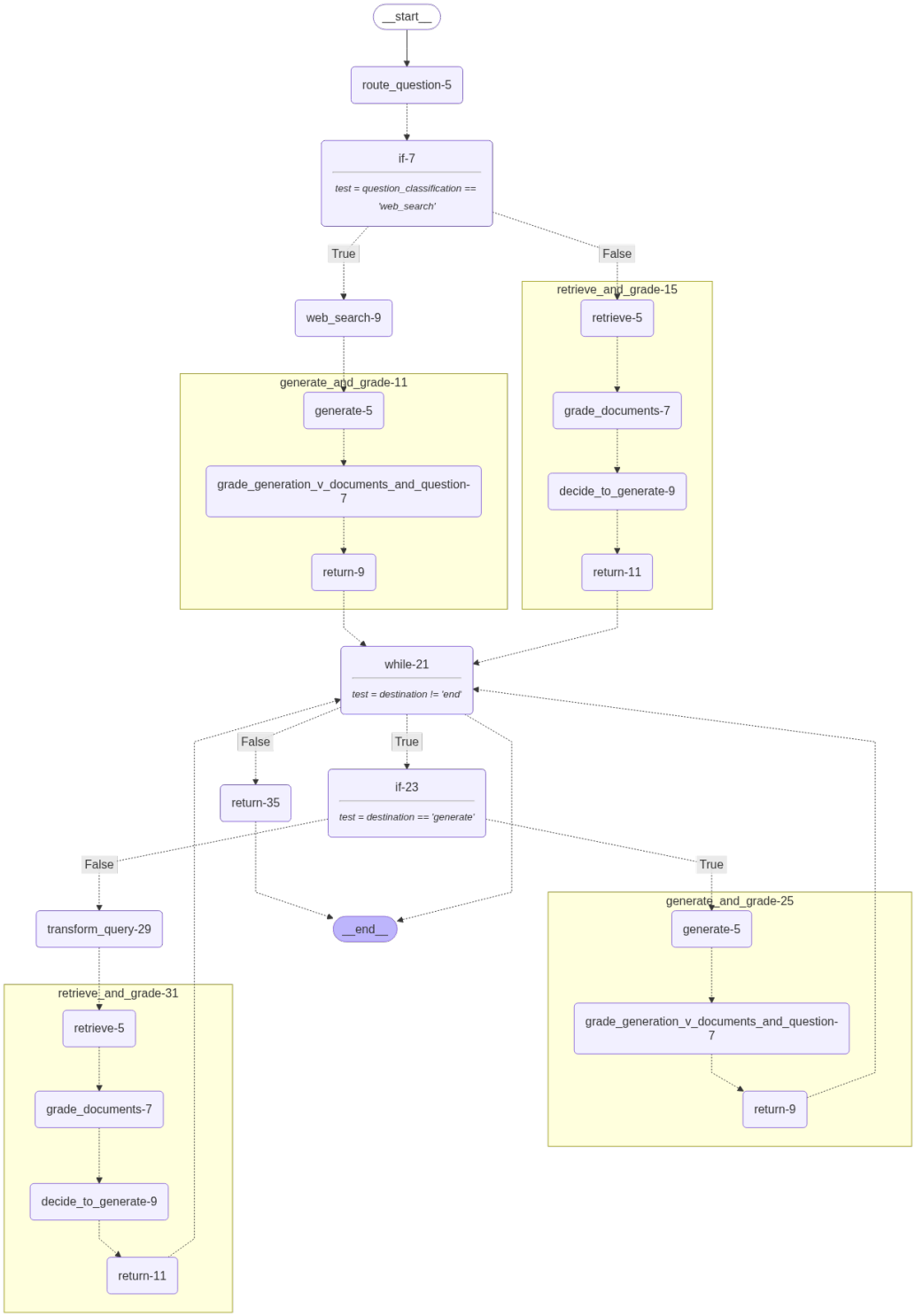

Figure 5.3 Graph generated for the MetaGraph approach to RAG

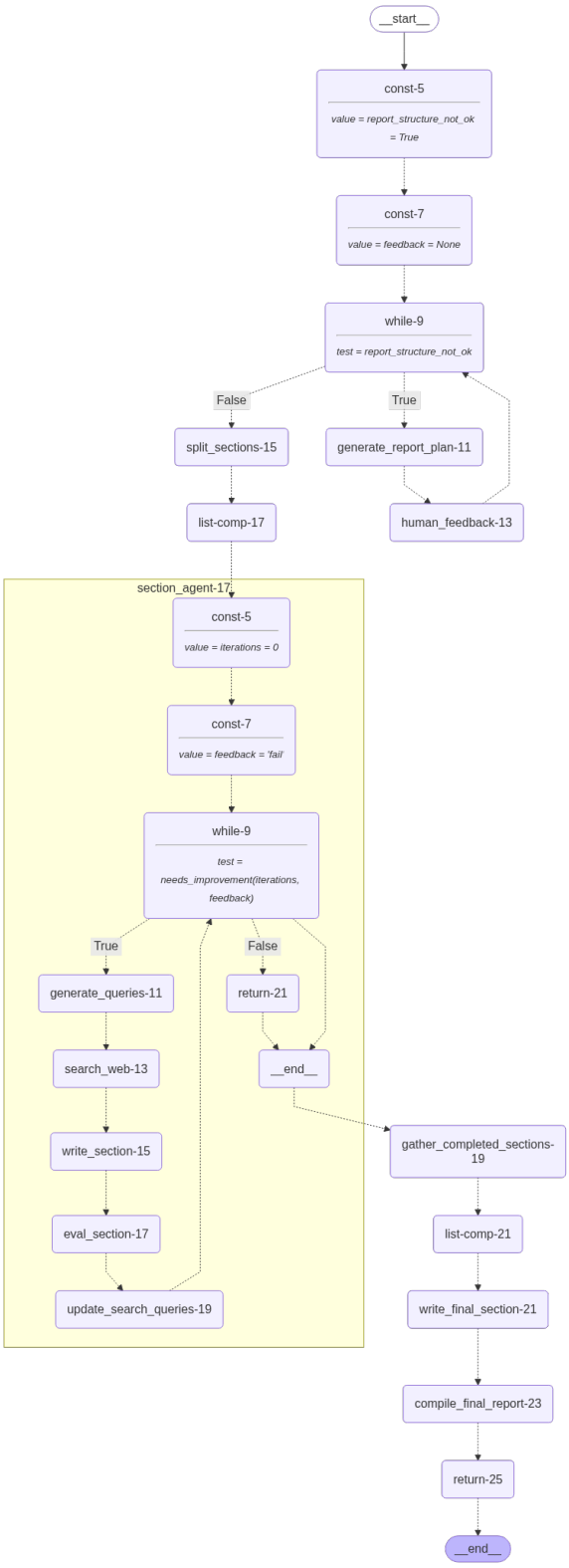

Figure 5.4 Graph generated for the MetaGraph approach to Open Deep Research

5.6 Limitations

As is demonstrated on the RAG example MetaGraph is not suitable for modeling workflows that are not linear in nature. That is however by design. The raison d'être of MetaGraph is to enable simple creation of graph-based workflows in LangGraph, not to create a completely new API for LangGraph that would match it in capabilities. It is possible to model such workflows using, as exemplified by the aforementioned RAG implementation, a go-to like (anti-)pattern[167].

Another possible concern is performance. For each node created MetaGraph takes the state, selects properties to pass as arguments to the function that is about to be called, calls the function and then creates a state update out of the call's result. This overhead is however completely insignificant when compared to how long is the typical execution of a created workflows. While MetaGraph may add milliseconds to the execution time a typical execution of Open Deep Research may take upwards of 10 minutes.

One possible disadvantage is that MetaGraph disables seemingly arbitrary functionalities of Python which makes it harder to use. This could be however mitigated by creating a MetaGraph linter or creating MetaGraph-specific rules for a Python linter such as ruff or PyLint.

Two biggest limitations are that not all Python features are supported and that is hard to use LangGraph graphs within MetaGraph (the other way is trivial). This could however be fixed with further development of this framework.

Conclusion

LLM-based agents are software whose control flow is at least partially steered by a LLM. There is a plethora of frameworks for their easier development. However their API tends to be hard to use.

The core idea of this thesis is that the best tool developers have to describe application logic is a programming language and that can be used to create agent within an already existing framework.

This thesis introduced MetaGraph, an AST rewriting generative metaprogramming framework. MetaGraph aims to ease the development of LLM-based agents within LangGraph framework by providing a way how develop them without having to use an unwieldy Graph API. It does so by taking Python functions as input and transforms them into LangGraph graphs which then can be used within the LangGraph framework.

This thesis has shown that this approach works well for some workflows while being unsuitable for others. The determining factor in that is the amount of non-linearity in the workflow. Workflows that are more akin to traditional procedural programming are easier to model while more graph-like workflows require the usage of goto-like antipatterns. To model these workflows easily a domain specific language would be required.

Bibliography

[1] Shervin Minaee et al. *Large Language Models: A Survey*. 2025. arXiv: 2402.06196 [cs.CL]. URL: https://arxiv.org/abs/2402.06196.

[2] C. E. Shannon. "Prediction and entropy of printed English". In: *The Bell System Technical Journal* 30.1 (1951), pp. 50–64. DOI: 10.1002/j.1538-7305.1951.tb01366.x.

[3] David E. Rumelhart and James L. McClelland. "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. 1987, pp. 318–362.

[4] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[5] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423/.

[6] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. arXiv: 1907.11692 [cs.CL]. URL: https://arxiv.org/abs/1907.11692.

[7] Zhenzhong Lan et al. *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations*. 2020. arXiv: 1909.11942 [cs.CL]. URL: https://arxiv.org/abs/1909.11942.

[8] Pengcheng He et al. *DeBERTa: Decoding-enhanced BERT with Disentangled Attention*. 2021. arXiv: 2006.03654 [cs.CL]. URL: https://arxiv.org/abs/2006.03654.

[9] Kevin Clark et al. "ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=r1xMH1BtvB.

[10] Alexis Conneau and Guillaume Lample. "Cross-lingual language model pretraining". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[11] Zhilin Yang et al. "XLNet: generalized autoregressive pretraining for language understanding". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[12] Li Dong et al. "Unified language model pre-training for natural language understanding and generation". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[13] Alec Radford et al. "Improving language understanding by generative pre-training". In: (2018).

[14] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: (2019).

[15] Colin Raffel et al. "Exploring the limits of transfer learning with a unified text-to-text transformer". In: *J. Mach. Learn. Res.* 21.1 (Jan. 2020). ISSN: 1532-4435.

[16] Linting Xue et al. "mT5: A Massively Multilingual Pre-trained Text-to-Text Transformer". In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Ed. by Kristina Toutanova et al. Online: Association for Computational Linguistics, June 2021, pp. 483–498. DOI: 10.18653/v1/2021.naacl-main.41. URL: https://aclanthology.org/2021.naacl-main.41/.

[17] Kaitao Song et al. "MASS: Masked Sequence to Sequence Pre-training for Language Generation". In: *ICML 2019*. June 2019. URL: https://www.microsoft.com/en-us/research/publication/mass-masked-sequence-to-sequence-pre-training-for-language-generation/.

[18] Mike Lewis et al. “BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Ed. by Dan Jurafsky et al. Online: Association for Computational Linguistics, July 2020, pp. 7871–7880. DOI: 10.18653/v1/2020.acl-main.703. URL: https://aclanthology.org/2020.acl-main.703/.

[19] Tom Brown et al. “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[20] Long Ouyang et al. “Training language models to follow instructions with human feedback”. In: *Proceedings of the 36th International Conference on Neural Information Processing Systems*. NIPS ’22. New Orleans, LA, USA: Curran Associates Inc., 2022. ISBN: 9781713871088.

[21] *Introducing ChatGPT — openai.com*. https://openai.com/index/chatgpt/. [Accessed 15-07-2025].

[22] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: 2303.08774 [cs.CL]. URL: https://arxiv.org/abs/2303.08774.

[23] Mark Chen et al. “Evaluating Large Language Models Trained on Code”. In: (2021). arXiv: 2107.03374 [cs.LG].

[24] Reiichiro Nakano et al. *WebGPT: Browser-assisted question-answering with human feedback*. 2022. arXiv: 2112.09332 [cs.CL]. URL: https://arxiv.org/abs/2112.09332.

[25] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: https://arxiv.org/abs/2302.13971.

[26] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL]. URL: https://arxiv.org/abs/2307.09288.

[27] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783.

[28] Rohan Taori et al. *Stanford Alpaca: An Instruction-following LLaMA model*. https://github.com/tatsu-lab/stanford_alpaca. 2023.

[29] Wei-Lin Chiang et al. *Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality*. Mar. 2023. URL: https://lmsys.org/blog/2023-03-30-vicuna/.

[30] Tim Dettmers et al. "QLORA: efficient finetuning of quantized LLMs". In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*. NIPS '23. New Orleans, LA, USA: Curran Associates Inc., 2023.

[31] Xinyang Geng et al. *Koala: A Dialogue Model for Academic Research*. Blog post. Apr. 2023. URL: https://bair.berkeley.edu/blog/2023/04/03/koala/ (visited on 04/03/2023).

[32] Albert Q. Jiang et al. *Mistral 7B*. 2023. arXiv: 2310.06825 [cs.CL]. URL: https://arxiv.org/abs/2310.06825.

[33] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. 2024. arXiv: 2308.12950 [cs.CL]. URL: https://arxiv.org/abs/2308.12950.

[34] Shishir G. Patil et al. *Gorilla: Large Language Model Connected with Massive APIs*. 2023. arXiv: 2305.15334 [cs.CL]. URL: https://arxiv.org/abs/2305.15334.

[35] Arka Pal et al. *Giraffe: Adventures in Expanding Context Lengths in LLMs*. 2023. arXiv: 2308.10882 [cs.AI]. URL: https://arxiv.org/abs/2308.10882.

[36] Aakanksha Chowdhery et al. "PaLM: scaling language modeling with pathways". In: *J. Mach. Learn. Res.* 24.1 (Jan. 2023). ISSN: 1532-4435.

[37] Paul Barham et al. *Pathways: Asynchronous Distributed Dataflow for ML*. 2022. arXiv: 2203.12533 [cs.DC]. URL: https://arxiv.org/abs/2203.12533.

[38] Yi Tay et al. "Transcending Scaling Laws with 0.1% Extra Compute". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 1471–1486. DOI: 10.18653/v1/2023.emnlp-main.91. URL: https://aclanthology.org/2023.emnlp-main.91/.

[39] Hyung Won Chung et al. *Scaling Instruction-Finetuned Language Models*. 2022. arXiv: 2210.11416 [cs.LG]. URL: https://arxiv.org/abs/2210.11416.

[40] Rohan Anil et al. *PaLM 2 Technical Report*. 2023. arXiv: 2305.10403 [cs.CL]. URL: https://arxiv.org/abs/2305.10403.

[41] Karan Singhal et al. “Large language models encode clinical knowledge”. In: *Nature* 620.7972 (Aug. 2023), pp. 172–180.

[42] Karan Singhal et al. “Toward expert-level medical question answering with large language models”. In: *Nature Medicine* 31.3 (Mar. 2025), pp. 943–950.

[43] Karan Singhal et al. *Large Language Models Encode Clinical Knowledge*. 2022. arXiv: 2212.13138 [cs.CL]. URL: https://arxiv.org/abs/2212.13138.

[44] Rico Sennrich, Barry Haddow, and Alexandra Birch. *Neural Machine Translation of Rare Words with Subword Units*. 2016. arXiv: 1508.07909 [cs.CL]. URL: https://arxiv.org/abs/1508.07909.

[45] Xinying Song et al. “Fast WordPiece Tokenization”. In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Ed. by Marie-Francine Moens et al. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 2089–2103. DOI: 10.18653/v1/2021.emnlp-main.160. URL: https://aclanthology.org/2021.emnlp-main.160/.

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

[47] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. “Self-Attention with Relative Position Representations”. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Ed. by Marilyn Walker, Heng Ji, and Amanda Stent. New Orleans, Louisiana: Association for Computational Linguistics, June 2018, pp. 464–468. DOI: 10.18653/v1/N18-2074. URL: https://aclanthology.org/N18-2074/.

[48] Jianlin Su et al. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 2023. arXiv: 2104.09864 [cs.CL]. URL: https://arxiv.org/abs/2104.09864.

[49] Ofir Press, Noah Smith, and Mike Lewis. “Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation”. In: *International Conference on Learning Representations*. 2022. URL: https://openreview.net/forum?id=R8sQPpGCv0.

[50] Noam Shazeer et al. “Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer”. In: *International Conference on Learning Representations*. 2017. URL: https://openreview.net/forum?id=B1ckMDqlg.

[51] Shengyu Zhang et al. *Instruction Tuning for Large Language Models: A Survey*. 2024. arXiv: 2308.10792 [cs.CL]. URL: https://arxiv.org/abs/2308.10792.

[52] Yizhong Wang et al. *Self-Instruct: Aligning Language Model with Self Generated Instructions*. 2022.

[53] Paul F. Christiano et al. “Deep reinforcement learning from human preferences”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS’17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 4302–4310. ISBN: 9781510860964.

[54] Rafael Rafailov et al. *Direct Preference Optimization: Your Language Model is Secretly a Reward Model*. 2024. arXiv: 2305.18290 [cs.LG]. URL: https://arxiv.org/abs/2305.18290.

[55] Kawin Ethayarajh et al. *KTO: Model Alignment as Prospect Theoretic Optimization*. 2024. arXiv: 2402.01306 [cs.LG]. URL: https://arxiv.org/abs/2402.01306.

[56] Ziwei Ji et al. “Survey of Hallucination in Natural Language Generation”. In: *ACM Comput. Surv.* 55.12 (Mar. 2023). ISSN: 0360-0300. DOI: 10.1145/3571730. URL: https://doi.org/10.1145/3571730.

[57] Kishore Papineni et al. “BLEU: a method for automatic evaluation of machine translation”. In: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. ACL ’02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 311–318. DOI: 10.3115/1073083.1073135. URL: https://doi.org/10.3115/1073083.1073135.

[58] Chin-Yew Lin. “ROUGE: A Package for Automatic Evaluation of Summaries”. In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: https://aclanthology.org/W04-1013/.

[59] Bhuwan Dhingra et al. *Handling Divergent Reference Texts when Evaluating Table-to-Text Generation*. 2019. arXiv: 1906.01081 [cs.CL]. URL: https://arxiv.org/abs/1906.01081.

[60] Haoyu Song et al. “Generating Persona Consistent Dialogues by Exploiting Natural Language Inference”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.05 (Apr. 2020), pp. 8878–8885. ISSN: 2159-5399. DOI: 10.1609/aaai.v34i05.6417. URL: http://dx.doi.org/10.1609/aaai.v34i05.6417.

[61] Or Honovich et al. *Q^2: Evaluating Factual Consistency in Knowledge-Grounded Dialogues via Question Generation and Question Answering*. 2021. arXiv: 2104.08202 [cs.CL]. URL: https://arxiv.org/abs/2104.08202.

[62] Nouha Dziri et al. *Evaluating Attribution in Dialogue Systems: The BEGIN Benchmark*. 2022. arXiv: 2105.00071 [cs.CL]. URL: https://arxiv.org/abs/2105.00071.

[63] Sashank Santhanam et al. *Rome was built in 1776: A Case Study on Factual Correctness in Knowledge-Grounded Response Generation*. 2022. arXiv: 2110.05456 [cs.CL]. URL: https://arxiv.org/abs/2110.05456.

[64] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.

[65] Shunyu Yao et al. *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*. 2023. arXiv: 2305.10601 [cs.CL]. URL: https://arxiv.org/abs/2305.10601.

[66] Potsawee Manakul, Adian Liusie, and Mark J. F. Gales. *SelfCheckGPT: Zero-Resource Black-Box Hallucination Detection for Generative Large Language Models*. 2023. arXiv: 2303.08896 [cs.CL]. URL: https://arxiv.org/abs/2303.08896.

[67] Noah Shinn et al. *Reflexion: Language Agents with Verbal Reinforcement Learning*. 2023. arXiv: 2303.11366 [cs.AI]. URL: https://arxiv.org/abs/2303.11366.

[68] Sarah J. Zhang et al. *Exploring the MIT Mathematics and EECS Curriculum Using Large Language Models*. 2023. arXiv: 2306.08997 [cs.CL]. URL: https://arxiv.org/abs/2306.08997.

[69] Tongshuang Wu et al. *PromptChainer: Chaining Large Language Model Prompts through Visual Programming*. 2022. arXiv: 2203.06566 [cs.HC]. URL: https://arxiv.org/abs/2203.06566.

[70] Yongchao Zhou et al. “Large Language Models are Human-Level Prompt Engineers”. In: *The Eleventh International Conference on Learning Representations*. 2023. URL: https://openreview.net/forum?id=92gvk82DE-.

[71] Yunfan Gao et al. *Retrieval-Augmented Generation for Large Language Models: A Survey*. 2024. arXiv: 2312.10997 [cs.CL]. URL: https://arxiv.org/abs/2312.10997.

[72] Timo Schick et al. "Toolformer: Language Models Can Teach Themselves to Use Tools". In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023. URL: https://openreview.net/forum?id=Yacmpz84TH.

[73] Markus Schlosser. "Agency". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2019. Metaphysics Research Lab, Stanford University, 2019.

[74] G.E.M. Anscombe. *Intention*. Library of philosophy and logic. Harvard University Press, 2000. ISBN: 9780674003996. URL: https://books.google.cz/books?id=_D1xjNXFT8cC.

[75] Uttam Mukhopadhyay et al. "An intelligent system for document retrieval in distributed office environments". In: *Journal of the American Society for Information Science* 37.3 (1986), pp. 123–135. DOI: https://doi.org/10.1002/(SICI)1097-4571(198605)37:3<123::AID-ASI3>3.0.CO;2-3. eprint: https://asistdl.onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291097-4571%28198605%2937%3A3%3C123%3A%3AAID-ASI3%3E3.0.CO%3B2-3. URL: https://asistdl.onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291097-4571%28198605%2937%3A3%3C123%3A%3AAID-ASI3%3E3.0.CO%3B2-3.

[76] "Intelligent agents: theory and practice". In: *The Knowledge Engineering Review* 10.2 (1995), pp. 115–152. DOI: 10.1017/S0269888900008122.

[77] Allen Newell and Herbert A. Simon. "Computer science as empirical inquiry: symbols and search". In: *Commun. ACM* 19.3 (Mar. 1976), pp. 113–126. ISSN: 0001-0782. DOI: 10.1145/360018.360022. URL: https://doi.org/10.1145/360018.360022.

[78] Rodney A. Brooks. "Intelligence Without Representation". In: *Artificial Intelligence* 47.1–3 (1991), pp. 139–159. DOI: 10.1016/0004-3702(91)90053-m.

[79] C. Ribeiro. "Reinforcement Learning Agents". In: *Artif. Intell. Rev.* 17.3 (May 2002), pp. 223–250. ISSN: 0269-2821. DOI: 10.1023/A:1015008417172. URL: https://doi.org/10.1023/A:1015008417172.

[80] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[81] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587 (Jan. 2016), pp. 484–489.

[82] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG]. URL: https://arxiv.org/abs/1312.5602.

[83] Jesse Farebrother, Marlos C. Machado, and Michael Bowling. *Generalization and Regularization in DQN*. 2019. URL: https://openreview.net/forum?id=HkGmDsR9YQ.

[84] Tim Brys et al. “Policy Transfer using Reward Shaping”. In: *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*. AAMAS ’15. Istanbul, Turkey: International Foundation for Autonomous Agents and Multiagent Systems, 2015, pp. 181–188. ISBN: 9781450334136.

[85] Chelsea Finn, Pieter Abbeel, and Sergey Levine. *Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks*. 2017. arXiv: 1703.03400 [cs.LG]. URL: https://arxiv.org/abs/1703.03400.

[86] Jason Wei et al. “Chain of Thought Prompting Elicits Reasoning in Large Language Models”. In: *Advances in Neural Information Processing Systems*. Ed. by Alice H. Oh et al. 2022. URL: https://openreview.net/forum?id=_VjQlMeSB_J.

[87] *Introduction to LLM Agents | NVIDIA Technical Blog — developer.nvidia.com*. https://developer.nvidia.com/blog/introduction-to-llm-agents/. [Accessed 14-07-2025].

[88] *What is an AI agent? — blog.langchain.com*. https://blog.langchain.com/what-is-an-agent/. [Accessed 15-07-2025].

[89] Lun Huang et al. *Attention on Attention for Image Captioning*. 2019. arXiv: 1908.06954 [cs.CV]. URL: https://arxiv.org/abs/1908.06954.

[90] Haotian Liu et al. “Visual Instruction Tuning”. In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh et al. Vol. 36. Curran Associates, Inc., 2023, pp. 34892–34916. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/6dcf277ea32ce3288914faf369fe6de0-Paper-Conference.pdf.

[91] Jiasen Lu et al. “UNIFIED-IO: A Unified Model for Vision, Language, and Multi-modal Tasks”. In: *The Eleventh International Conference on Learning Representations*. 2023. URL: https://openreview.net/forum?id=E01k9048soZ.

[92] Junnan Li et al. *BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models*. 2023. arXiv: 2301.12597 [cs.CV]. URL: https://arxiv.org/abs/2301.12597.

[93] Rongjie Huang et al. "AudioGPT: understanding and generating speech, music, sound, and talking head". In: *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*. AAAI'24/IAAI'24/EAAI'24. AAAI Press, 2024. ISBN: 978-1-57735-887-9. DOI: 10.1609/aaai.v38i21.30570. URL: https://doi.org/10.1609/aaai.v38i21.30570.

[94] Yuan Gong, Yu-An Chung, and James Glass. "AST: Audio Spectrogram Transformer". In: *Proc. Interspeech 2021*. 2021, pp. 571–575. DOI: 10.21437/Interspeech.2021-698.

[95] John R. Searle. "What is language: some preliminary remarks". In: *John Searle's Philosophy of Language: Force, Meaning and Mind*. Ed. by Savas L.Editor Tsohatzidis. Cambridge University Press, 2007, pp. 15–46.

[96] Zhuosheng Zhang and Hai Zhao. *Advances in Multi-turn Dialogue Comprehension: A Survey*. 2021. arXiv: 2103.03125 [cs.CL]. URL: https://arxiv.org/abs/2103.03125.

[97] Yejin Bang et al. *A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity*. 2023. arXiv: 2302.04023 [cs.CL]. URL: https://arxiv.org/abs/2302.04023.

[98] Alec Radford et al. *Better Language Models and Their Implications*. OpenAI Blog. Blog post announcing GPT-2 — "Better language models and their implications". Feb. 2019. URL: https://openai.com/index/better-language-models/.

[99] R. Thomas McCoy et al. "How Much Do Language Models Copy From Their Training Data? Evaluating Linguistic Novelty in Text Generation Using RAVEN". In: *Transactions of the Association for Computational Linguistics* 11 (June 2023), pp. 652–670. ISSN: 2307-387X. DOI: 10.1162/tacl_a_00567. eprint: https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00567/2141008/tacl_a_00567.pdf. URL: https://doi.org/10.1162/tacl%5C_a%5C_00567.

[100] Stefanie Tellex et al. "Understanding Natural Language Commands for Robotic Navigation and Mobile Manipulation". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 25.1 (Aug. 2011), pp. 1507–1514. DOI: 10.1609/aaai.v25i1.7979. URL: https://ojs.aaai.org/index.php/AAAI/article/view/7979.

[101] Jessy Lin et al. “Inferring Rewards from Language in Context”. In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Smaranda Muresan, Preslav Nakov, and Aline Villavicencio. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 8546–8560. DOI: 10.18653/v1/2022.acl-long.585. URL: https://aclanthology.org/2022.acl-long.585/.

[102] Hong Jun Jeon, Smitha Milli, and Anca D. Dragan. *Reward-rational (implicit) choice: A unifying formalism for reward learning*. 2020. arXiv: 2002.04833 [cs.LG]. URL: https://arxiv.org/abs/2002.04833.

[103] Manzil Zaheer et al. *Big Bird: Transformers for Longer Sequences*. 2021. arXiv: 2007.14062 [cs.LG]. URL: https://arxiv.org/abs/2007.14062.

[104] Amirkeivan Mohtashami and Martin Jaggi. *Landmark Attention: Random-Access Infinite Context Length for Transformers*. 2023. arXiv: 2305.16300 [cs.CL]. URL: https://arxiv.org/abs/2305.16300.

[105] Yuxiang Nie et al. *Capturing Global Structural Information in Long Document Question Answering with Compressive Graph Selector Network*. 2022. arXiv: 2210.05499 [cs.CL]. URL: https://arxiv.org/abs/2210.05499.

[106] Bing Wang et al. *SCM: Enhancing Large Language Model with Self-Controlled Memory Framework*. 2025. arXiv: 2304.13343 [cs.CL]. URL: https://arxiv.org/abs/2304.13343.

[107] Joon Sung Park et al. *Generative Agents: Interactive Simulacra of Human Behavior*. 2023. arXiv: 2304.03442 [cs.HC]. URL: https://arxiv.org/abs/2304.03442.

[108] Wanjun Zhong et al. *MemoryBank: Enhancing Large Language Models with Long-Term Memory*. 2023. arXiv: 2305.10250 [cs.CL]. URL: https://arxiv.org/abs/2305.10250.

[109] Ali Modarressi et al. *RET-LLM: Towards a General Read-Write Memory for Large Language Models*. 2024. arXiv: 2305.14322 [cs.CL]. URL: https://arxiv.org/abs/2305.14322.

[110] Ziheng Huang et al. *Memory Sandbox: Transparent and Interactive Memory Management for Conversational Agents*. 2023. arXiv: 2308.01542 [cs.HC]. URL: https://arxiv.org/abs/2308.01542.

[111] Chenxu Hu et al. *ChatDB: Augmenting LLMs with Databases as Their Symbolic Memory*. 2023. arXiv: 2306.03901 [cs.AI]. URL: https://arxiv.org/abs/2306.03901.

[112] Jared Kaplan et al. *Scaling Laws for Neural Language Models*. 2020. arXiv: 2001.08361 [cs.LG]. URL: https://arxiv.org/abs/2001.08361.

[113] Ivan Vulić et al. "Probing Pretrained Language Models for Lexical Semantics". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Ed. by Bonnie Webber et al. Online: Association for Computational Linguistics, Nov. 2020, pp. 7222–7240. DOI: 10.18653/v1/2020.emnlp-main.586. URL: https://aclanthology.org/2020.emnlp-main.586/.

[114] Tara Safavi and Danai Koutra. *Relational World Knowledge Representation in Contextual Language Models: A Review*. 2021. arXiv: 2104.05837 [cs.CL]. URL: https://arxiv.org/abs/2104.05837.

[115] Badr AlKhamissi et al. *A Review on Language Models as Knowledge Bases*. 2022. arXiv: 2204.06031 [cs.CL]. URL: https://arxiv.org/abs/2204.06031.

[116] Ronald Kemker et al. "Measuring catastrophic forgetting in neural networks". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*. AAAI'18/IAAI'18/EAAI'18. New Orleans, Louisiana, USA: AAAI Press, 2018. ISBN: 978-1-57735-800-8.

[117] Nicola De Cao, Wilker Aziz, and Ivan Titov. *Editing Factual Knowledge in Language Models*. 2021. arXiv: 2104.08164 [cs.CL]. URL: https://arxiv.org/abs/2104.08164.

[118] Freda Shi et al. *Large Language Models Can Be Easily Distracted by Irrelevant Context*. 2023. arXiv: 2302.00093 [cs.CL]. URL: https://arxiv.org/abs/2302.00093.

[119] Jie Huang and Kevin Chen-Chuan Chang. *Towards Reasoning in Large Language Models: A Survey*. 2023. arXiv: 2212.10403 [cs.CL]. URL: https://arxiv.org/abs/2212.10403.

[120] Jason Wei et al. "Emergent Abilities of Large Language Models". In: *Transactions on Machine Learning Research* (2022). Survey Certification. ISSN: 2835-8856. URL: https://openreview.net/forum?id=yzkSU5zdwD.

[121] Xuezhi Wang et al. *Self-Consistency Improves Chain of Thought Reasoning in Language Models*. 2023. arXiv: 2203.11171 [cs.CL]. URL: https://arxiv.org/abs/2203.11171.

[122] Zhiheng Xi et al. “Self-Polish: Enhance Reasoning in Large Language Models via Problem Refinement”. In: *Findings of the Association for Computational Linguistics: EMNLP 2023*. Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 11383–11406. DOI: 10.18653/v1/2023.findings-emnlp.762. URL: https://aclanthology.org/2023.findings-emnlp.762/.

[123] Aman Madaan et al. *Self-Refine: Iterative Refinement with Self-Feedback*. 2023. arXiv: 2303.17651 [cs.CL].

[124] Antonia Creswell, Murray Shanahan, and Irina Higgins. “Selection-Inference: Exploiting Large Language Models for Interpretable Logical Reasoning”. In: *The Eleventh International Conference on Learning Representations*. 2023. URL: https://openreview.net/forum?id=3Pf3Wg6o-A4.

[125] Laura Sebastia, Eva Onaindia, and Eliseo Marzal. “Decomposition of planning problems”. In: *AI Communications* 19.1 (2006), pp. 49–81. DOI: 10.3233/EAI-2006-361. eprint: https://journals.sagepub.com/doi/pdf/10.3233/EAI-2006-361. URL: https://journals.sagepub.com/doi/abs/10.3233/EAI-2006-361.

[126] Denny Zhou et al. “Least-to-Most Prompting Enables Complex Reasoning in Large Language Models”. In: *The Eleventh International Conference on Learning Representations*. 2023. URL: https://openreview.net/forum?id=WZH7099tgfM.

[127] Yue Wu et al. *Plan, Eliminate, and Track – Language Models are Good Teachers for Embodied Agents*. 2023. arXiv: 2305.02412 [cs.CL]. URL: https://arxiv.org/abs/2305.02412.

[128] Guohao Li et al. “CAMEL: communicative agents for ”mind” exploration of large language model society”. In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*. NIPS ’23. New Orleans, LA, USA: Curran Associates Inc., 2023.

[129] Shunyu Yao et al. *ReAct: Synergizing Reasoning and Acting in Language Models*. 2023. arXiv: 2210.03629 [cs.CL]. URL: https://arxiv.org/abs/2210.03629.

[130] Sébastien Bubeck et al. *Sparks of Artificial General Intelligence: Early experiments with GPT-4*. 2023. arXiv: 2303.12712 [cs.CL]. URL: https://arxiv.org/abs/2303.12712.

[131] Irene Solaiman and Christy Dennison. *Process for Adapting Language Models to Society (PALMS) with Values-Targeted Datasets*. 2021. arXiv: 2106.10328 [cs.CL]. URL: https://arxiv.org/abs/2106.10328.

[132] Yujia Qin et al. *Tool Learning with Foundation Models*. 2024. arXiv: 2304.08354 [cs.CL]. URL: https://arxiv.org/abs/2304.08354.

[133] Nicholas Carlini et al. "Extracting training data from diffusion models". In: *Proceedings of the 32nd USENIX Conference on Security Symposium*. SEC '23. Anaheim, CA, USA: USENIX Association, 2023. ISBN: 978-1-939133-37-3.

[134] Stephen Roller et al. "Recipes for Building an Open-Domain Chatbot". In: *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*. Ed. by Paola Merlo, Jorg Tiedemann, and Reut Tsarfaty. Online: Association for Computational Linguistics, Apr. 2021, pp. 300–325. DOI: 10.18653/v1/2021.eacl-main.24. URL: https://aclanthology.org/2021.eacl-main.24/.

[135] Chen Ling et al. *Domain Specialization as the Key to Make Large Language Models Disruptive: A Comprehensive Survey*. 2024. arXiv: 2305.18703 [cs.CL]. URL: https://arxiv.org/abs/2305.18703.

[136] Aaron Parisi, Yao Zhao, and Noah Fiedel. *TALM: Tool Augmented Language Models*. 2022. arXiv: 2205.12255 [cs.CL]. URL: https://arxiv.org/abs/2205.12255.

[137] Tianle Cai et al. "Large Language Models as Tool Makers". In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=qV83K9d5WB.

[138] Yingqiang Ge et al. "OpenAGI: When LLM Meets Domain Experts". In: *In Advances in Neural Information Processing Systems (NeurIPS)* (2023).

[139] Andres M Bran et al. *ChemCrow: Augmenting large-language models with chemistry tools*. 2023. arXiv: 2304.05376 [physics.chem-ph]. URL: https://arxiv.org/abs/2304.05376.

[140] Jingqing Ruan et al. *TPTU: Large Language Model-based AI Agents for Task Planning and Tool Usage*. 2023. arXiv: 2308.03427 [cs.AI]. URL: https://arxiv.org/abs/2308.03427.

[141] Yannis Lilis and Anthony Savidis. "A Survey of Metaprogramming Languages". In: *ACM Comput. Surv.* 52.6 (Oct. 2019). ISSN: 0360-0300. DOI: 10.1145/3354584. URL: https://doi.org/10.1145/3354584.

[142] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[143] Guy L. Steele. *Common LISP: the language (2nd ed.)* USA: Digital Press, 1990. ISBN: 1555580416.

[144] R. Kent Dybvig. *The Scheme Programming Language, 4th Edition*. 4th. The MIT Press, 2009. ISBN: 026251298X.

[145] Gregor Kiczales et al. "An Overview of AspectJ". In: *ECOOP 2001 — Object-Oriented Programming*. Ed. by Jørgen Lindskov Knudsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 327–354. ISBN: 978-3-540-45337-6.

[146] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. "AspectC++: an aspect-oriented extension to the C++ programming language". In: *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*. CRPIT '02. Sydney, Australia: Australian Computer Society, Inc., 2002, pp. 53–60. ISBN: 0909925887.

[147] *Aspect Oriented Programming with Spring :: Spring Framework*. URL: https://docs.spring.io/spring-framework/reference/core/aop.html (visited on 07/17/2025).

[148] V. Subramaniam. *Programming Groovy 2: Dynamic Productivity for the Java Developer*. Pragmatic programmers Bd. 2. Pragmatic Bookshelf, 2013. ISBN: 9781937785307. URL: https://books.google.cz/books?id=jvbTmAEACAAJ.

[149] Steven Diamond and Stephen Boyd. *CVXPY: A Python-Embedded Modeling Language for Convex Optimization*. 2016. arXiv: 1603.00943 [math.OC]. URL: https://arxiv.org/abs/1603.00943.

[150] D. Batory, B. Lofaso, and Y. Smaragdakis. "JTS: Tools for Implementing Domain-Specific Languages". In: *Proceedings of the 5th International Conference on Software Reuse*. ICSR '98. USA: IEEE Computer Society, 1998, p. 143. ISBN: 0818683775.

[151] Shan Shan Huang, David Zook, and Yannis Smaragdakis. "cJ: enhancing java with safe type conditions". In: *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*. AOSD '07. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2007, pp. 185–198. ISBN: 1595936157. DOI: 10.1145/1218563.1218584. URL: https://doi.org/10.1145/1218563.1218584.

[152] Dirk Draheim, Christof Lutteroth, and Gerald Weber. "A Type System for Reflective Program Generators". In: *Generative Programming and Component Engineering*. Ed. by Robert Glück and Michael Lowry. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 327–341. ISBN: 978-3-540-31977-1.

[153] Shan Shan Huang, David Zook, and Yannis Smaragdakis. "Statically Safe Program Generation with SafeGen". In: *Generative Programming and Component Engineering*. Ed. by Robert Glück and Michael Lowry. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 309–326. ISBN: 978-3-540-31977-1.

[154] DAVIDE ANCONA and ELENA ZUCCA. "A theory of mixin modules: basic and derived operators". In: *Mathematical Structures in Computer Science* 8.4 (1998), pp. 401–446. DOI: 10.1017/S0960129598002576.

[155] Nathanael Schärli et al. "Traits: Composable Units of Behaviour". In: *ECOOP 2003 – Object-Oriented Programming*. Ed. by Luca Cardelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 248–274. ISBN: 978-3-540-45070-2.

[156] John Reppy and Aaron Turon. "Metaprogramming with Traits". In: *ECOOP 2007 – Object-Oriented Programming*. Ed. by Erik Ernst. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 373–398. ISBN: 978-3-540-73589-2.

[157] Walid Taha and Tim Sheard. "Multi-stage programming with explicit annotations". In: *SIGPLAN Not.* 32.12 (Dec. 1997), pp. 203–217. ISSN: 0362-1340. DOI: 10.1145/258994.259019. URL: https://doi.org/10.1145/258994.259019.

[158] Cristiano Calcagno et al. "Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection". In: *Generative Programming and Component Engineering*. Ed. by Frank Pfenning and Yannis Smaragdakis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 57–76. ISBN: 978-3-540-39815-8.

[159] Qingyun Wu et al. "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation". In: *COLM 2024*. Aug. 2024. URL: https://www.microsoft.com/en-us/research/publication/autogen-enabling-next-gen-llm-applications-via-multi-agent-conversation-framework/.

[160] Victor Dibia et al. "AutoGen Studio: A No-Code Developer Tool for Building and Debugging Multi-Agent Systems". Preprint. Aug. 2024. URL: https://www.microsoft.com/en-us/research/publication/autogen-studio-a-no-code-developer-tool-for-building-and-debugging-multi-agent-systems/.

[161] *LangChain — langchain.com*. https://www.langchain.com/. [Accessed 14-07-2025].

[162] *Introduction | LangChain — python.langchain.com*. https://python.langchain.com/docs/introduction/. [Accessed 14-07-2025].

[163] Grzegorz Malewicz et al. “Pregel: a system for large-scale graph processing”. In: *Proceedings of the 2010 international conference on Management of data*. New York, NY, USA, 2010, pp. 135–146. URL: http://doi.acm.org/10.1145/1807167.1807184.

[164] K. Sveidqvist and A. Jain. *The Official Guide to Mermaid.js: Create complex diagrams and beautiful flowcharts easily using text and code*. Packt Publishing, 2021. ISBN: 9781801076258. URL: https://books.google.cz/books?id=CBQ-EAAAQBAJ.

[165] Soyeong Jeong et al. *Adaptive-RAG: Learning to Adapt Retrieval-Augmented Large Language Models through Question Complexity*. 2024. arXiv: 2403.14403 [cs.CL]. URL: https://arxiv.org/abs/2403.14403.

[166] OpenAI. *Introducing Deep Research*. Accessed: 2025-07-16. Feb. 2025. URL: https://openai.com/index/introducing-deep-research/.

[167] Edsger W. Dijkstra. “Letters to the editor: go to statement considered harmful”. In: *Commun. ACM* 11.3 (Mar. 1968), pp. 147–148. ISSN: 0001-0782. DOI: 10.1145/362929.362947. URL: https://doi.org/10.1145/362929.362947.