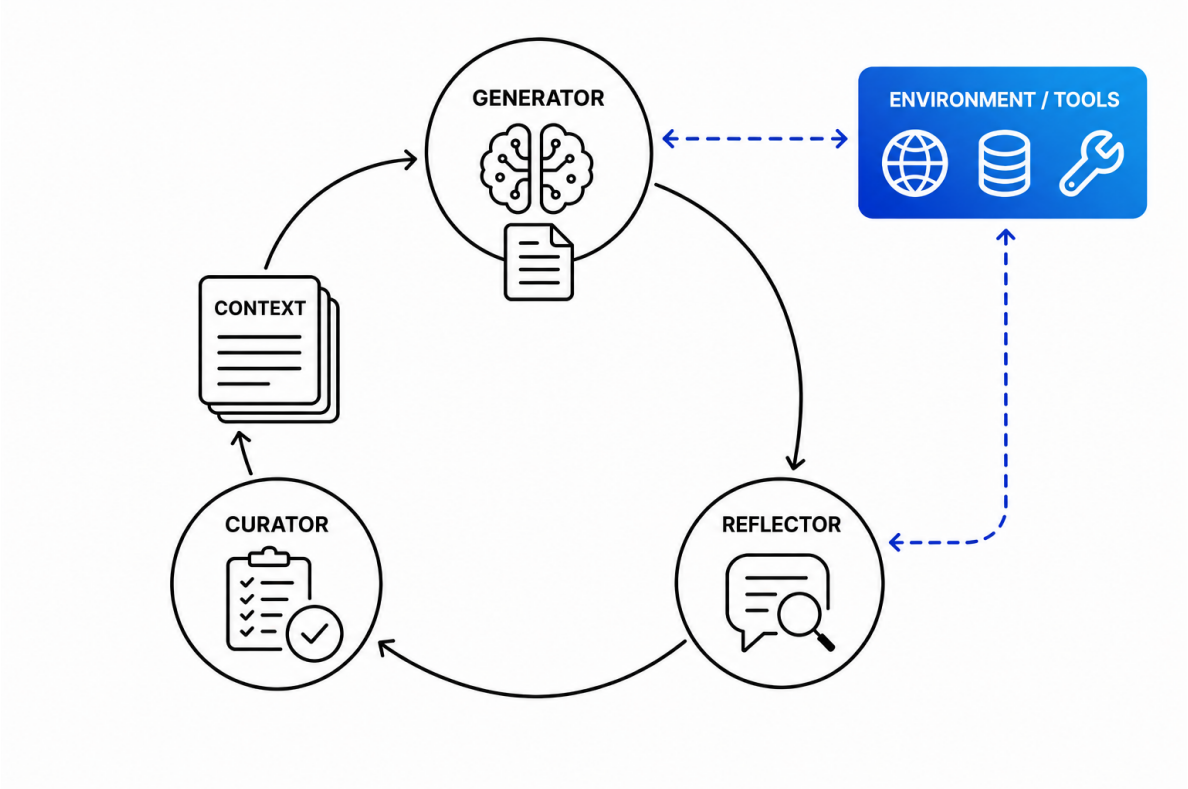

Self-Improving LLM Contexts for Agentic AI

Enhancing AI Agent Performance Through Agentic Context Engineering

Master's thesis in Data Science and AI

WILLIAM FRISK
ISAC HANSSON

DEPARTMENT OF MATHEMATICAL SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Master's thesis 2026

Self-Improving LLM Contexts for Agentic AI

Enhancing AI Agent Performance Through Agentic Context Engineering

WILLIAM FRISK
ISAC HANSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mathematical Sciences
Chalmers University of Technology
Gothenburg, Sweden 2026

Self-Improving LLM Contexts for Agentic AI
Enhancing AI Agent Performance Through Agentic Context Engineering
WILLIAM FRISK
ISAC HANSSON

Supervisor: Aron Lagerberg, Recorded Future
Examiner: Johan Jonasson, Department of Mathematical Sciences

Master's Thesis 2026
Department of Mathematical Sciences
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A minimalist flow diagram illustrating the iterative interaction between the generator, reflector, curator and environment within an agentic context engineering framework. (Created with ChatGPT, 2026)

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2026

Self-Improving LLM Contexts for Agentic AI
Enhancing AI Agent Performance Through Agentic Context Engineering
WILLIAM FRISK
ISAC HANSSON
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

Large language model (LLM) agents have become increasingly capable systems for solving complex tasks through reasoning, tool use, and interaction with external environments. Recent work has shown that the LLM agents can be improved not only through parameter fine-tuning, but also through modifying the context. This thesis investigates context adaptation for domain-specific LLM agents in a threat intelligence environment. In collaboration with Recorded Future, we implement and evaluate a framework based on Agentic Context Engineering (ACE) for optimizing a search-oriented LLM agent through iterative generation, reflection and curation. The framework is further extended with ReAct-style reflection and the incorporation of external domain documentation during optimization. The approach is evaluated on datasets constructed from realistic threat intelligence tasks and compared against baseline agents with either minimal or handcrafted context. Experimental results show that the context adaptation substantially improves agent performance, achieving higher search accuracy and grounding. The best performing framework, with the new contributions and without ground truth supervision, achieves **15.1** percentage points increase in pass@1. Overall, the findings demonstrate that context engineering is an effective approach for adapting LLM agents to specialized enterprise domains. Furthermore, indication is given that ground truth supervision might be substitutable for environment interaction.

Keywords: Agentic context engineering (ACE), large language models (LLM), AI agents, context adaptation, prompt optimization.

Acknowledgements

We would like to express our gratitude to Aron Lagerberg at Recorded Future for defining the project scope, providing valuable support throughout our work, and creating a welcoming environment. Further, we would like to thank Recorded Future for providing the tools and equipment necessary to complete this work. We are also deeply grateful to our academic supervisor and examiner, Johan Jonasson, for his continuous guidance and rapid feedback.

William Frisk and Isac Hansson, Gothenburg, May, 2026

List of Acronyms

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

ACE	Agentic Context Engineering
AI	Artificial Intelligence
API	Application Programming Interface
CoT	Chain of Thought Prompting
CRM	Customer Relationship Management
CTF	Capture-the-flag
GEPA	Genetic Pareto
GPU	Graphics Processing Unit
GRPO	Group Relative Policy Optimization
ICL	In-Context Learning
LLM	Large Language Model
MCP	Model Context Protocol
ML	Machine Learning
MSE	Mean Squared Error
NLP	Natural Language Processing
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SFT	Supervised Fine-tuning
TQL	Term Query Language

Contents

List of Figures

List of Tables

1

Introduction

Artificial Intelligence (AI) is increasingly integrated into everyday life for both industry professionals and the general public. Its impacts span a range of domains, be that increased productivity, lost jobs, or digital persons to chat with. McKinsey & Company [1] describes AI as "(...) the largest organizational paradigm shift since the industrial and digital revolutions." From an academic standpoint, this motivates further investigation into how AI systems can be improved, with focus on AI agents, i.e. systems capable of interacting with an environment such as a code base. These agents represent one of the most rapidly developing applications of AI. This thesis therefore investigates how AI agents can be optimized to improve performance in industrial settings.

AI broadly refers to systems that are able to reason in order to perform tasks, whether through rule-based systems or modern neural network architectures. An AI agent is formed by coupling this reasoning capability with perception and the ability to act within an environment [2]. Such an agent maintains memory of previous states and knowledge of the environment, receives observations from it, and takes actions in order to achieve specific goals. AI agents in practice range from self-driving cars to autonomous trading systems.

The development of AI agents has been accelerated by the emergence of large language models (LLMs). In isolation, LLMs function as next-token[1] prediction models that achieve strong performance on tasks such as question answering and text summarization. Their capabilities have led to widespread adoption and significant public interest, exemplified by ChatGPT reaching one million users within five days of release [3].

AI agents can utilize LLMs as a central reasoning component, enabling more autonomous behavior. These systems extend the capabilities of LLMs by introducing tools, reasoning, information retrieval and multi-step planning. When providing LLMs with these capabilities, they can be used to perform substantially complex tasks. AI agents built on LLMs are commonly referred to as LLM agents.

[1] A token is a basic unit of text, e.g. a word or a punctuation mark. Tokens and tokenization are explained in greater detail in Chapter 2.

An example of an LLM agent is Claudius. In an experiment conducted by Anthropic in collaboration with Andon Labs, an LLM-based agent named Claudius was deployed to manage a simulated vending machine business [4], [5]. Claudius was equipped with a set of tools enabling it to operate the business autonomously. These included web search, email-based procurement, customer relationship management (CRM) for tracking customers, note-taking capabilities, form creation and processing, and an internal messaging system for customer communication. With access to these capabilities, Claudius was able to plan, order, and sell products in order to maintain business operations.

Claudius and other LLM agents have become possible due to the rapid advancement of LLM capabilities in recent years [6]. A significant factor behind this progress has been research focused on model architecture improvements and the increasing number of parameters in these models. Over the past three years, there has been an exponential increase in the number of parameters in state-of-the-art LLMs, as illustrated in Figure 1.1. One of the reasons for this has been an increase in the computational power of hardware.

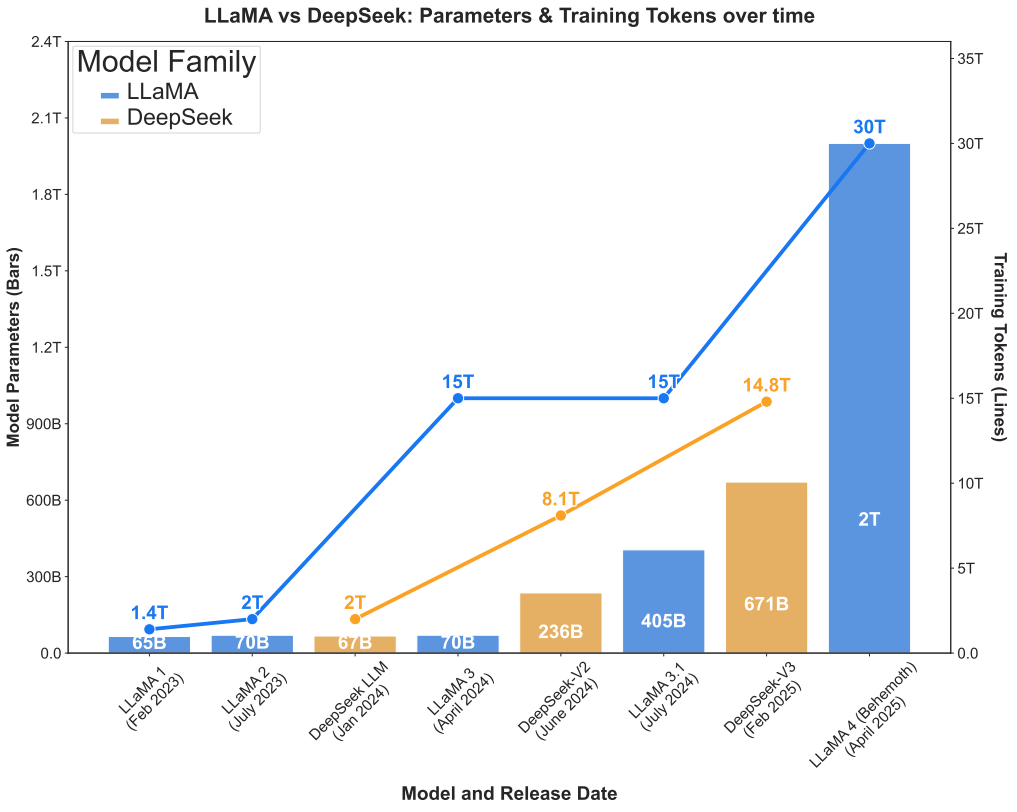

Figure 1.1: Bar graph showing the exponential increase in the number of model parameters in the largest state-of-the-art LLMs over time [7], [8], [9], [10], [11], [12], [13]. Two line graphs are overlayed showing the amount of training data in tokens for the models.

Increases in model size and architectural improvements have led to substantial performance gains. However, further improvements in specific domains, particularly for LLM agents, are often achieved through a process known as fine-tuning. Fine-tuning refers to the adjustment of a model's parameters, typically through small updates, in order to improve performance on specific tasks without degrading its general capabilities [14].

One common approach is supervised fine-tuning (SFT), in which the model is trained on correct "ground truth" outputs. SFT is often used to incorporate domain-specific knowledge or to shape how the model should behave and respond in particular contexts. However, in many cases, especially in LLM agent settings, ground truth outputs are not readily available. In such situations, reinforcement learning (RL) can be applied. RL generally involves defining a reward function that assigns higher values to desirable outputs and lower values to undesirable ones. The model is then trained to maximize this reward signal through iterative updates to its parameters.

Recent research, and the focus of this thesis, explores alternative approaches to improving domain-specific and agentic performance in LLMs, in particular by performing learning at the application layer. This refers to modifications of the text and prompts provided as input to the model [15], [16], [17]. The collection of input information supplied to an LLM is referred to as its context, and the model generates outputs conditioned on this context. The maximum amount of information that can be processed at once is known as the context window. As shown in Figure 1.2, context window sizes have increased over time, enabling models to incorporate more information during generation. As a result, there has been a shift from learning primarily through parameter updates (e.g., SFT and RL) toward methods that instead modify and optimize information within the context. This paradigm is commonly referred to as context adaptation.

Context adaptation can leverage a model's internal knowledge and reasoning patterns. Empirical studies have shown that carefully designed prompts and prompt-tuning methods can match or, in some cases, outperform traditional reinforcement learning approaches on domain-specific and agentic tasks. Examples of such prompt-based approaches include ReAct and GEPA [16], [19].

Beyond potential performance improvements over traditional reinforcement learning, there is also strong motivation for context adaptation due to the closed-source nature of many commercial LLMs. In such models, access to parameters and training data is restricted, which limits the feasibility of fine-tuning or reinforcement learning-based adaptation that requires direct parameter updates. Context adaptation avoids this limitation by operating entirely at the application layer, as described above. This makes it particularly relevant in settings where models are accessed only via application programming interfaces (APIs)[2] or where organizations rely on proprietary LLMs that cannot be directly modified.

[2]Application programming interfaces (APIs) define structured interfaces through which software components can interact. They specify how a system exposes functionality to other software, allowing external programs to access services, data, or operations in a standardized way. APIs can

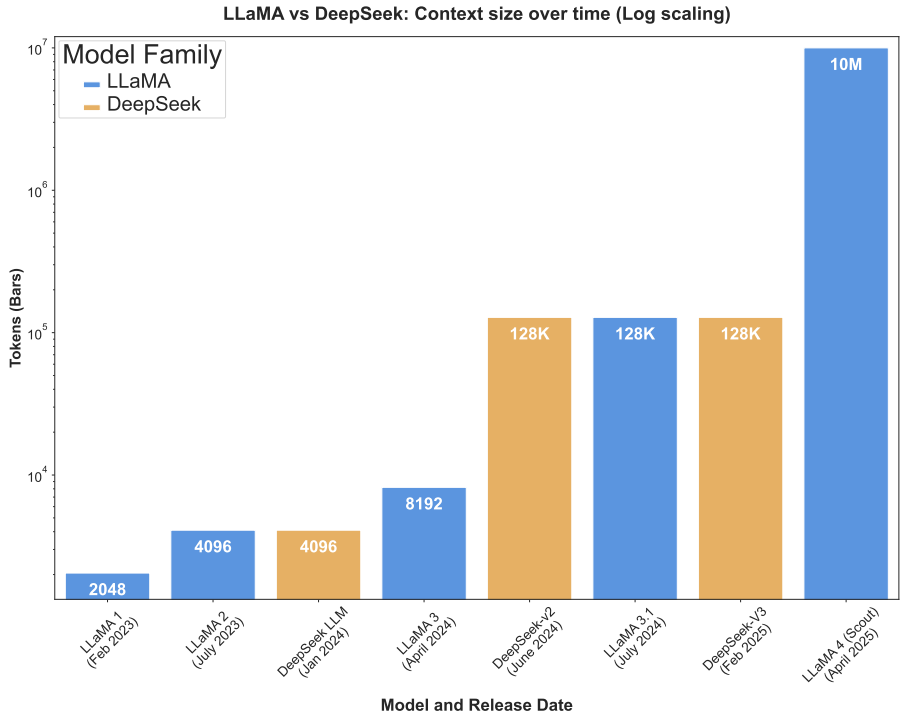

Figure 1.2: Bar graph showing the increase of the context window size in state-of-the-art LLMs over time [9], [10], [11], [12], [13], [18]. The graph has logarithmic scaling vertically.

Another motivation for context adaptation is cost, both in terms of monetary expense and environmental impact. In the setting of fine-tuning LLMs for task-solving, the term "rollout" is commonly used to describe a complete attempt at solving a task, from the initial prompt to the final output. The authors of GEPA report that their context adaptation method required, on average, only 16% of the rollouts needed for traditional reinforcement learning to achieve comparable performance across all but one benchmark [16]. The authors of ACE, another context adaptation framework, report a further reduction of rollout requirements by 75.1% relative to GEPA [15]. Since fewer rollouts generally correspond to fewer generated tokens, this translates into lower computational cost and reduced environmental impact.

The exact monetary and energy savings achieved through context adaptation frameworks are difficult to estimate due to differences between methods and the limited availability of energy consumption data from LLM providers. However, context adaptation methods such as ACE generally require significantly fewer rollouts compared to traditional reinforcement learning approaches [15], [16]. Since each rollout involves repeated LLM text generation, reducing the number of rollouts can substantially decrease both computational cost and energy consumption during optimization.

be local, such as library interfaces, or remote, such as web APIs used for communication between distributed systems.

The potential applications of LLM agents, together with the effectiveness of context adaptation, motivate the work presented in this thesis. In collaboration with Recorded Future[3], we implement and evaluate context engineering methods applied to an LLM agent designed for answering queries about cyber threats. This is done to answer the question: how effective is Agentic Context Engineering (ACE) for optimizing an LLM agent in a closed-source industry environment?

The implementation is based on the framework proposed by Q. Zhang et al., titled Agentic Context Engineering (ACE) [15], and is extended with additional design choices, including the incorporation of external documentation into the reflection process and the introduction of a ReAct-enabled reflector. The framework is trained and optimized using data covering common use cases of the agent. It is evaluated against two baseline LLM agents: one without substantial contextual information, and one representing Recorded Future's current production system, which uses manually engineered context. The evaluation measures task completion accuracy and hallucination rate and further includes metrics such as token usage and latency.

The results demonstrate that context adaptation can substantially improve the performance of LLM agents in realistic settings. The optimized agents consistently outperform the baseline without structured context and, in most cases, also outperform the agent using manually engineered context. For the strongest configurations, improvements of just over 15 percentage points in pass@1 are observed.

The results further indicate that the contributions of this thesis, namely the ReAct-enabled reflector and access to domain-specific documentation, account for the largest performance gains. Notably, the ReAct-enabled reflector achieves competitive performance even without reliance on ground-truth data during training.

Overall, these findings suggest that context adaptation is an effective approach for adapting commercial and closed-source LLMs to niche use cases, thereby increasing their practical value for organizations and their customers.

1.1 The Use of LLMs

This work focuses on developing a framework for context adaptation in LLM agents. As such, the experiments use LLMs for the evaluation of the proposed methods. For report writing, we used LLMs only to correct style, language and grammar. LLMs were not used to generate new text from scratch.

[3]Recorded Future is a cybersecurity and threat intelligence company that aggregates large-scale data from the open web, dark web, technical sources, and internal telemetry. It applies machine learning and natural language processing to analyze this data and surface actionable intelligence [20]. Its core function is to support organizations in detecting, prioritizing, and responding to cyber threats, vulnerabilities, and emerging risks in real time.

2

Large Language Models

The foundation to the methods in this thesis is LLMs. They are machine learning models that are trained to generate text that adheres to the rules of language. This section presents some machine learning fundamentals, building up to the domain of language modeling and finally the design of large language models.

2.1 Machine Learning Fundamentals

Traditional programming requires the software engineer to write precise and detailed instructions for the computer to follow. This works well in many situations but in some cases, where the rules are complex or flexible, writing a program for the machine to follow can be time-consuming and prohibitively difficult. Consider programming a computer to predict the sale price of a house. While a real estate expert can intuitively estimate the price by looking at its condition, location etc., specifying exactly in code how to weigh hundreds of variables, like the quality of nearby schools, roofing material or the neighborhood, is incredibly difficult.

Machine learning (ML) is a subset of the field of artificial intelligence (AI) that consists of building systems that learn patterns from data to make predictions or decisions without explicit programming. Using ML one can design a model which, provided data, can learn to solve a variety of different tasks, for example forecasting house sale prices, without a human having to program the explicit mathematical conditions for every possible feature contributing to the sale price.

Teaching an ML model, most often called training, entails adapting the model to solving a specific problem. This is done by updating the values of the models parameters which determine the output predictions. Calculating how to change the parameters can be done in several ways but the most simple and common way is so called supervised learning. Supervised learning is a technique that uses labeled data to train the ML model [21]. Labeled datasets consist of data points that are paired with their respective correct output which is commonly called the *ground truth*. Supervised learning works by minimizing the difference between what the ML model predicts as output and what the ground truth output was. This difference is expressed in the loss function which in its most simple form is the mean squared error (MSE) as seen in Equation (2.1), where n is the size of the dataset, y_i is the

ground truth and $\hat{y}_i$ is the predicted output of the ML model. The parameters of the ML model which determine the prediction are usually updated by performing an algorithm from the gradient descent family to minimize the value of the loss function.

$$\text{MSE} := \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.1}$$

Since the value of the loss function is trivially related to the accuracy of the predictions, with a correctly designed ML model and correctly labeled data, minimizing the loss over several iterations of gradient descent one will get a more and more accurate ML model. This is what the learning in machine learning means, i.e. iteratively minimizing the error between predicted outputs and correct outputs.

2.1.1 Learning in Neural Networks

Neural networks are one of the more influential types of ML models in modern AI. In broad terms they consist of stacks of simple "neurons" in layers that learn weights and biases from the data [22]. Neural networks are inspired by biological neurons found in human brains and their main idea is that the combination of many simple parts comprise a complex whole.

In mathematical terms a neural network learns some function $f(X)$ by taking a vector of features $X = (x_1, x_2, x_3, \dots)$ as input and outputting a prediction vector Y [22]. Like mentioned before it does this by combining layers of neurons. Each neuron computes its output (activation), a, according to Equation (2.2), where x_i are the input features, w_i are learned weights, b a learned bias and $\sigma(\cdot)$ is some activation function. Individually these neurons are very simple and basically a linear regression. However, when combined in layers with neurons feeding their output into other neurons their expressibility becomes much greater.

$$a = \sigma(\sum_{i=1}^{n} w_i x_i + b) \tag{2.2}$$

The activation function is typically a nonlinear function that is used to extend the neural networks ability to learn to not only linear relationships but also more complex ones. Examples of commonly used activation functions can be seen in Equation (2.3).

$$\begin{aligned} \text{Sigmoid} &:= \frac{1}{1+e^{-x}} \\ \text{ReLU} &:= \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases} \\ \text{Tanh} &:= \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \end{aligned} \tag{2.3}$$

A neural network typically consists of three types of layers, the input layer, hidden layers and an output layer, as seen in Figure 2.1 [22]. The input layer holds the raw features as seen in the data, $(x_1, x_2, x_3 \dots)$. The hidden layers consist of neurons that transform the inputs into new representations. The output layer consists of a nonlinear activation function which transforms the learned representation produced by the neurons into a result that is used for the final prediction, such as a number or a probability distribution.

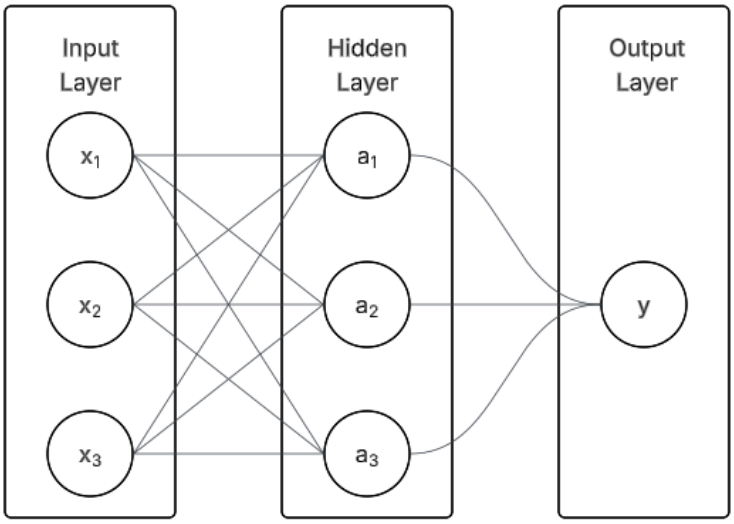

Figure 2.1: Example of small neural network with input, hidden and output layer.

2.1.1.1 Backpropagation

As touched upon above, an ML model learns by minimizing the value of a loss function. In neural networks this is done using an algorithm called backpropagation [22]. The backpropagation algorithm consists of four steps:

1. Forward pass: Inputs consisting of data points in the training dataset are fed into the network which computes the linear combination, nonlinear activation and final output prediction.

2. Loss calculation: The loss function measures how correct or incorrect the predictions were as compared to the ground truth.

3. Backward pass: The loss is propagated backwards through the neural network and at each neuron the algorithm calculates how much each weight and bias contributed to the loss using the chain rule.

4. Weight update: The weights and biases are updated slightly to reduce the loss using a gradient descent method.

These four steps are repeated many times over the dataset and each iteration helps the network update the weights and biases to reduce the loss and improve its accuracy.

In order to illustrate how this works in mathematical terms, consider the simple neural network in Figure 2.1. Say one wanted to calculate how much to update w_1 which determines how much x_1 affects the activation a_1, this would mean having to calculate $\frac{\partial \text{loss}}{\partial w_1}$. This can be solved using the chain rule which gives that $\frac{\partial \text{loss}}{\partial w_1} = \frac{\partial \text{loss}}{\partial y}\frac{\partial y}{\partial a_1}\frac{\partial a_1}{\partial w_1}$. This gradient is then used to update w_1 using some gradient descent algorithm, which is exemplified in Equation (2.4), where α is some small number called step size.

$$w_1 \leftarrow w_1 - \alpha \cdot \frac{\partial \text{loss}}{\partial w_1} \tag{2.4}$$

2.1.2 Classification Problems

Problems in ML that are of the type where one wants to assign a discrete label or class to the input are called classification problems. A common example of a classification problem is determining whether an image depicts a dog or a cat.

In classification one usually uses a softmax activation function in the output layer of the neural network, as seen in Equation (2.5). This function takes the representation from the hidden layers as input and outputs a probability distribution over all possible classes $\{1, \ldots, K\}$.

$$\text{Softmax}(x_i) := \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{2.5}$$

The aim of an ML model applied to a classification problem is minimizing the categorical cross-entropy loss, Equation (2.6), where K is the number of classes y_j is the ground truth label with 1 if class j is the correct class and 0 otherwise and $\hat{y}_j$ is the predicted probability of class j.

$$\text{CE} := -\sum_{j=1}^{K} y_j \log(\hat{y}_j) \tag{2.6}$$

Neural networks can easily be applied to classification problems by using the softmax activation function in the output layer and minimizing the categorical cross-entropy loss using backpropagation.

2.2 Language Modeling

In ML, language modeling is the task of constructing a probability distribution function that assigns a probability to every string in a language [23]. The higher the probability of a string, the more likely that it is a valid construction in the language. In more broad terms, language modelling is the task of, given an input string, predicting what the continuation of that string is.

Predicting the continuation of a string is, in essence, a classification problem where the set of possible classes is all possible continuations to the string. This means that neural networks can be applied to this task to try to learn this probability function.

2.2.1 Tokenization

To limit the amount of possible continuations to a finite set of classes one usually divides language into a set of tokens that come one after the other. This set of tokens is called a vocabulary. In its most simple form a token is one word. Splitting language up this way, the ML model can take a sequence of tokens, words, as input and output a single token according to the predicted distribution conditioned on the input.

In practice tokenization methods are more complex than just splitting on each word. This is because the amount of words in a language can be very large which makes it inefficient to learn for the model. For example, if every distinct English word were represented as its own token, the vocabulary could easily contain hundreds of thousands or even millions of entries when including compound words, inflections, names, and rare terms. Instead one usually keeps shorter words as a single token, like “dog” while splitting longer words like “darkness” into “dark” and “ness” [24]. This way the “ness” token can be reused in other words like “brightness.” OpenAI states that for common English text, a rule of thumb of thumb is that one token is approximately 4 characters [25].

2.2.2 Recurrent Neural Networks

Language is inherently sequential in the sense that the order of the tokens matter. Traditional neural networks have a hard time modeling and learning relationships in sequences. As a result of this other neural network architectures have been invented. One of the more influential architectures are recurrent neural networks (RNN).

To understand a sentence you must remember the words that came before the current word, RNNs model this kind of memory in a so called hidden state which acts as a form of internal memory [26]. An RNN processes sequences one element at a time. At each time step t, it takes the current token x_t and the memory from the previous step h_{t-1} to produce a new hidden state h_t. Mathematically, the update is often expressed as seen in Equation (2.7), where σ is some activation function.

$$h_t = \sigma(W_h x_t + U_h h_{t-1} + b_h) \tag{2.7}$$

The hidden state, h_t, captures the grammar and meaning of the phrase so far. The model uses the final hidden state to generate a probability distribution over the vocabulary to predict the next token.

2.3 Transformers

Transformers are a type of neural network that, like RNNs, were designed to handle sequences of input data. The transformer neural network architecture was introduced in 2017 by a team at Google and has been by far the most effective architecture for language modeling [27].

The transformer neural network architecture is the foundation of all modern LLMs. It enables modeling of distant relationships in sequences while still being relatively computationally efficient. The core components of the transformer are the attention mechanism, encoder and decoder.

2.3.1 Attention Mechanism

The attention mechanism is inspired by the ability of humans to pay special attention to important details while ignoring others [28]. In this vein, the attention mechanism is an ML technique that allows ML models to attend to the most relevant parts of the input data.

The attention mechanism consists of three core processes [28]. Firstly, reading the input sequences and converting each element in the sequence to an embedding vector, which is a learned dense vector representation of a token in a continuous vector space. Tokens with similar semantic or syntactic meaning are typically mapped to nearby representations in this space. After mapping the tokens to their vector representations, the vectors are compared to determine how similar two embedding vectors in the sequence are and this is used to compute a weight in the interval $[0, 1]$. A weight of 0 means that the element should be ignored and a weight of 1 means that it should receive all of the attention. Finally, a process of using those attention weights to influence how the model makes its predictions.

Vaswani et al. [27] define the process of attention as an interaction between three types of vectors for each token in a sequence. First, the query vector, q, which

represents the information which a given token is seeking [28]. Second, the key vectors, k, which represent the information each token contains. Third, the value vectors, v, which contain the actual information of each input in the sequence. All these vectors, q, k and v, are linear transformations of the embedding vectors of the input tokens and they are attained by multiplying the embedding vector of an input token with matrices of learned parameters for each type of vector, W_q, W_k, W_v, see Equation (2.8).

$$\begin{aligned} q_i &= W_q x_i \\ k_i &= W_k x_i \\ v_i &= W_v x_i \end{aligned} \tag{2.8}$$

To compute what tokens in the sequence the current token should pay attention to Vaswani et al. [27] use a mechanism called scaled dot-product attention. The first step is computing the dot-product between the q vector of the current token and the k vector of all other tokens in the sequence, as seen in (2.9), where d_k is the dimensionality of k and the $\frac{1}{\sqrt{d_k}}$ term is present for numerical stability.

$$s_{i,j} = \frac{q_i^\top k_j}{\sqrt{d_k}} \tag{2.9}$$

These are then mapped to attention scores between 0 and 1 using softmax. The attention scores, α, are then used to weight the impact of the v vectors, see Equation (2.10), where T_s is the sequence length and c_t is the resulting vector that summarizes the most relevant information from the input sequence.

$$\begin{aligned} \alpha_{i,j} &= \text{softmax}(s_{i,j}) \\ c_t &= \sum_{i=1}^{T_s} \alpha_{i,j} v_i \end{aligned} \tag{2.10}$$

When generating the next token, the transformer uses c_t to bias the generation based on the most relevant information summarized by the attention. In this way it works similar to the hidden state, h_{t-1}, present in RNNs.

Using the attention mechanism, the transformer is able to learn what tokens to attend to and not when predicting the next token. In practice this could be exemplified by the transformer learning that in the sentence "The animal didn't cross the street because it was too tired," the word *it* refers to the animal and not the street and using that information in future token generation.

2.3.2 Encoder-Decoder

The attention layers are combined with standard neural networks into stacks. In [27] the transformer architecture is described as having two different types of stacks, an encoder and a decoder. The encoder is responsible for taking an input sequence and embedding all relevant information from that sequence into a vector space. The decoder on the other hand is responsible for generating the output probabilities for the language model, conditioned on the embedding vectors of the encoder.

The decoder can be used auto-regressively, meaning that its output probability can be sampled and the resulting token can be fed into the decoder to continue generating an output sequence. In fact, the decoder and encoder can be used completely on their own, decoupled from the other. It is common to use only the decoder part to generate language that completes some input sequence. These types of decoder-only transformers are the most common types of transformers in LLMs.

2.4 Extending Foundation Models

Transformers and LLMs are very powerful neural networks but they require a vast amount of training data, computational resources and research to be effective. For example, the model presented by Vaswani et al. [27] is trained on eight GPUs for 3.5 days and that model is very small compared to state-of-the-art LLMs. The Llama 3.2 model with 3 billion parameters created by Meta was trained for approximately 460 000 GPU hours [29]. That means approximately 55 years of continuous training if the training was not done in parallel on multiple GPUs.

As a result of the vast amounts of time and data it takes to train state-of-the-art LLMs it has become extremely common to use pretrained LLMs as a base for applications and then adapt them through transfer learning or prompting for specific use cases. These pretrained LLMs are commonly called foundation models and they are provided by a range of companies like OpenAI with their line of GPT models or Meta and their herd of Llama models.

2.4.1 Transfer Learning

One way of adapting a foundation model to a specific use case is through transfer learning. Transfer learning involves continuing the training through updating the learned parameters of the LLM on a smaller dataset. This is usually done through fine-tuning which entails updating the parameters of the LLM through backpropagation and gradient descent like normal but only updating the weights a very small amount and using the pretrained weights as a base to start from. This way the LLM can be adapted to a specific use case, like answering queries about legal cases, without disrupting the underlying knowledge about language which the LLM has learned during its pretraining.

2.4.2 Prompting

Another way of adapting a foundation model is to present it with specific prompts. Through the use of clear prompts which the LLM can attend to as its input sequence it can generate its output in specific ways. A naive example would be to input a law book in the prompt to allow the model to attend to the details there and through that learn how to answer legal queries. More details on how adapting an LLM through prompts can be done are presented in Chapter 4.

3

LLM Agents

With some of the fundamentals behind language modeling, transformers and LLMs presented in the previous chapter, this chapter aims to build on this by describing how LLMs are used as the brain in AI agents. First a short introduction to the components of LLM applications is given which builds to LLM agent taxonomy and finally domain adaptation and evaluation.

3.1 Components of LLM Applications

Most modern LLM applications are built around a small set of core components. At a high level, a user provides an input prompt which, together with additional contextual information, is sent to the LLM. The model then generates a response conditioned on this input.

Typically, the input to the LLM consists of the three main parts seen in Figure 3.1: the system prompt, the conversation history, and the current user message [30]. The system prompt contains high-level instructions that guide the behavior of the model, such as defining its role, constraints, or desired response style. The conversation history contains previous messages exchanged within the same session and provides contextual information to maintain coherence across multiple turns. Finally, the current user message contains the latest query or instruction.

These components are assembled into a single input context for each request. The LLM then processes this context and generates an output [31].

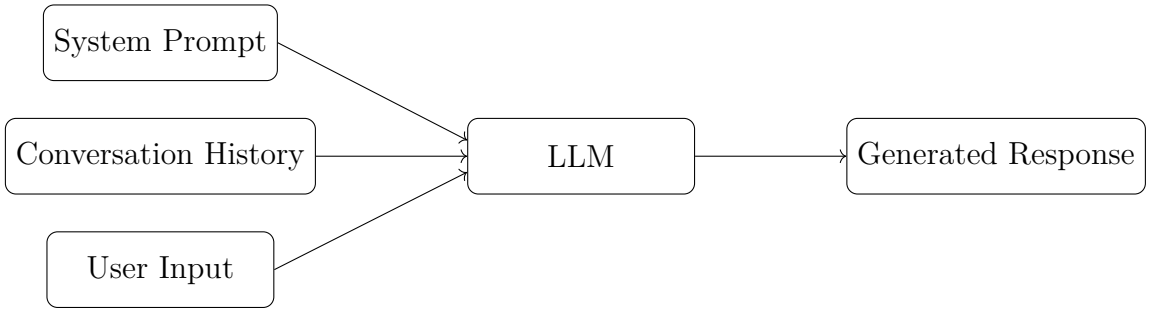

Figure 3.1: Simplified architecture of a typical LLM application.

The context is a collective term for everything the LLM takes as input. As mentioned above, this includes the system prompt, conversation history, and user input. However, the context may also contain additional information used by the LLM to generate its output, such as tool descriptions, outputs returned by external tools, and skills, which are described later in this chapter.

3.2 LLMs in AI Agents

AI agents are generalized by Qu et al. as "autonomous systems capable of perceiving their surroundings, reasoning about possible courses of action, and executing decisions" [32, p. 1]. Poole and Mackworth provide an even more general definition of an agent, "an agent is something that acts in an environment" [2, ch. 2.1]. Early AI agents relied on hand-crafted rules and their capability was limited. However, their power has increased greatly with time [32]. Contemporary AI agents are deployed in settings such as self-driving cars, automated trading in financial markets and code generation, analysis and execution.

A major leap in the performance and capabilities of AI agents has come as a result of the invention of LLMs [33]. In modern LLM agents, LLMs do not only generate text but act as decision making components that can reason about observations, plan for goals and choose actions. In a taxonomy presented by Luo et al. [33], LLM agents consist of four core components: profile definition, memory mechanisms, planning capability and action execution.

The first of these components is the agent profile which defines the agents identity, behavior and role in the system [33]. This profile can be manually specified by the developer, for example by prompting it to be a researcher, programmer or planner. However, it may also be dynamically generated depending on the task or environment. The second component is memory. Luo et al. distinguish between short-term memory, which stores the current dialogue history, reasoning and environment feedback and long-term memory which stores reusable knowledge across tasks. Even so, the profile and short- and long-term memory is text that is incorporated into the prompt of the LLM. The LLM then uses this input text to condition its output on, thereby steering the behavior of the LLM agent.

The third component of Luo et al. taxonomy is planning [33]. Rather than attempting to solve a complex task in a single step, LLM agents can decompose the problem into smaller subtasks and solve them sequentially. The fourth component and final component is action execution, which is what connects the reasoning with the environment in which the LLM agent acts. The main form of action execution is tool use. Tool use enables the LLM agent to perform things the LLM cannot reliably do on its own, such as performing exact computations, retrieving up-to-date information or executing code.

3.3 Agent Tools

Tools are a critical component of modern LLM agents. While LLMs provide strong reasoning capabilities, they are limited by their training data and the basic functionality of generating text. Tools extend the LLM agents capability beyond simple text generation [34].

Tools are executable functions that an LLM can invoke to perform specific actions like performing arithmetic or getting todays weather [34]. However, like mentioned before LLMs can only receive text inputs and generate text outputs, meaning they have no inherent way to call tools. The way this is solved is that the LLM is told that it has access to tools in its context and if it decides that it needs to call a tool it generates an output that represents a tool call, say "call add(5,2)". The system around the LLM is then responsible for interpreting this and calling the tool and then inputting the result back into the LLM for the next text generation.

3.3.1 MCP

The tools that an agent has access to are called via a standardized protocol called model context protocol (MCP). MCP follows a client-server architecture where the host application, i.e., the LLM agent, contains one or more MCP clients used to communicate with MCP servers [35]. MCP servers act as wrappers around APIs, databases and other software or data sources. They retrieve information and return it to the MCP clients as context that the LLM can interpret.

MCP servers expose descriptions of their capabilities and what parameters are necessary for the tool call. This metadata is fetched by the MCP clients and injected into the context of the LLM before generation. This means that the LLM knows what tools it has available, their capabilities and what arguments it needs to pass to them to function. Figure 3.2 shows what steps an MCP tool call consists of and in what order.

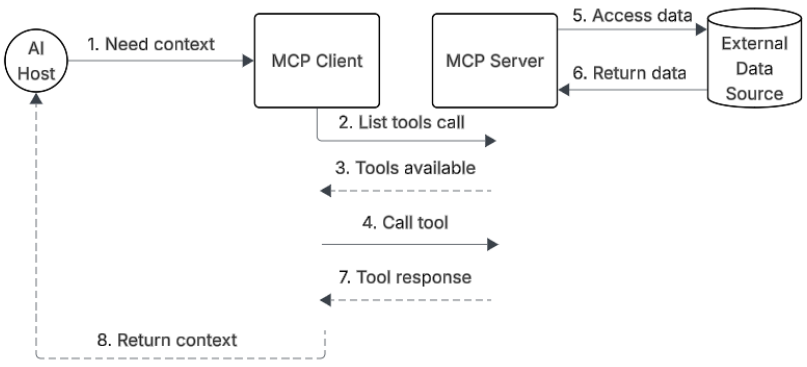

Figure 3.2: Diagram showing the steps of an MCP tool call.

3.4 Agent Skills

While MCP tools extend an agent's capabilities by allowing it to interact with external systems, tools alone are often insufficient for solving complex tasks reliably. In many cases, an agent also requires structured guidance describing *how* tasks should be approached. This introduces the problem of managing large amounts of reusable instructions and contextual knowledge within the limited context window of the LLM. Referring back to the taxonomy of LLM agents presented in Section 3.2, agent skills can be viewed as a mechanism for structuring and retrieving such knowledge. Skills provide reusable task-specific instructions, workflows, and resources that can be dynamically loaded into the LLM context when needed.

Storing information on how to solve tasks in the context provided to an agent is crucial. It has been shown many times that this improves performance of AI agents [15], [17]. However, not every instruction that one wants to provide to the agent in general is pertinent to a specific task. Further the context window size of an LLM is limited, longer contexts increases token costs of prompting and even though the context window of the largest LLMs have reached millions of tokens the attention mechanisms can have difficulties in processing long contexts [36].

Agent Skills reduces the amount of information that needs to be stored in the LLM context window [37]. Anthropic proposes a framework for organizing folders of instructions, scripts and other resources that agents can discover and load dynamically. Each of these resources are provided through a markdown file that contains a YAML preamble containing a name and a description and then markdown formatted instructions explaining the skill, see Figure 3.3. Besides explicit instructions a skill can also link to other skills through providing the paths to the respective markdown file.

The end result is a scheme of progressive disclosure which at the first level contains the name and description of all skills at the first level [37]. Then through linking more and more layers of detail can be uncovered by the agent. The first level of names and descriptions are incorporated into the agents context at startup and all deeper levels are removed from the context in between queries.

The loading of the skills and their linked files into the LLM context is most commonly done through an MCP server. By calling different tools on this server, the LLM agent can load skills and linked files depending on what it deems necessary for the task at hand.

3.5 Fine-Tuning LLM Agents

To adapt the LLM agent to act a certain way and improve performance in certain domains, one can apply parameter fine-tuning of the LLM inside the agent. However, constructing ground truth outputs to use for supervised learning is difficult in the agentic setting because tool interactions and solutions tend to be long. Therefore

```
---
name: summarize-document
description: Guidelines for condensing a long
   document into a concise summary highlighting key
   points and important details.
---

# Summarizing Documents
These guidelines should be followed when condensing
   long documents into concise summaries.

## Inputs
- `document_text` (string): The full text to
   summarize
- `max_length` (integer, optional): Desired summary
   length

## Outputs
- `summary` (string): A clear, structured summary of
   the document

## Behavior
1. Identify the main themes and key arguments in the
   text
2. Remove redundant or low-importance details
3. Produce a coherent summary within the desired
   length
4. Preserve factual accuracy and original meaning

## Example Usage
- **Input:** A 2,000-word article on climate policy
- **Output:** A 150-word summary covering the main
   policy proposals and implications
```

Figure 3.3: Example of skill file with YAML header and markdown body.

reinforcement learning (RL) is commonly employed as an alternative. RL works by defining a reward function that outputs higher values for good actions and lower values for bad actions. The aim is then to maximize the expected value of this reward function.

Reinforcement learning can be employed in the LLM agent setting. One intuitive method is called group relative policy optimization (GRPO). On a high level, without introducing a lot of mathematical notation, GRPO works by letting the LLM generate a group of outputs to a question and then calculating a reward by comparing these outputs [38]. The aim is to maximize the expected reward given the LLM parameters and the gradient estimates to do this are backpropagated through the LLM to update its parameters.

3.6 Evaluation of LLM Agents

When an LLM agent has been designed, implemented, fine-tuned etc. it needs to be evaluated to verify performance. However, one of the main complexities of implementing LLM agents is constructing an automatic and unbiased evaluation pipeline. The capabilities that make agents useful, i.e. their complexity, ability to adapt and nondeterministic nature is also exactly what makes them hard to evaluate. Asking the same question of an LLM agent multiple times might give different outputs, several of which could be correct. Moreover, there might be several different paths to reach the same final output. Further, since outputs are free text it is hard to define a static metric for whats wrong and whats right. The exact metrics and definition of right or wrong will depend on the application. However, there are some more or less established methods and design philosophies for how to evaluate LLM agents.

3.6.1 pass@k and pass^k

LLMs are in themselves deterministic, in the sense that given the same input they produce the same output distribution of the probability of the next token. To enhance the creativity of LLMs and allow for more diverse outputs one most often introduces some randomness by sampling from this output distribution. Because of this, the AI agents that build upon LLMs become nondeterministic as well. This is great for making the agents more creative and reduce repetition. It does, on the other hand, introduce some complexity to the evaluation. The same input does not necessarily generate the same output every time and hence providing the agent with the same test case prompt may lead to a passing test some times and a failing test other times. To deal with this uncertainty accuracy metrics for LLM agents are usually expressed in terms of pass@k (pass at k) and pass^k (pass hat k) [39].

Consider, for now, some kind of generic test function that takes as input the output of the LLM agent and outputs a boolean value representing whether or not the output satisfies some criterion. The pass@k metric measures the probability that, out of k attempts, at least one solution passes this test [40]. This can be calculated

by running some number of trials and reporting the fraction of k-tuples of trials that succeeded at least once. However, this usually exhibits high variance and takes a lot of computation if one wants to examine different k. So in practice one usually generates $n \geq k$ samples and calculates an unbiased estimate for pass@k as seen in Equation (3.1), where c is the amount of samples that passes the test and the expectation is over the dataset of tasks presented to the LLM agent.

$$\text{pass@}k := \mathbb{E}_{\text{tasks}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{3.1}$$

In contrast to pass@k, pass^k is defined as the probability that *all* k attempts pass the test [41]. pass^k was introduced for real-world agent tasks that require reliability and consistency. Similarly to pass@k this could be computed by running some number of trials and reporting the fraction of k-tuples where all k trials succeeded. But for the same reasons as above, in practice one usually calculates an unbiased estimator as seen in Equation (3.2).

$$\text{pass}\hat{}k := \mathbb{E}_{\text{tasks}} \left[\frac{\binom{c}{k}}{\binom{n}{k}} \right] \tag{3.2}$$

3.6.2 LLM-as-judge

With the nondeterministic nature of AI agents dealt with through pass@k and pass^k the question still remains of defining the test function, i.e. how one decides if a task has been solved or not. This is not trivial, firstly because of the free text answers which often lead to synonyms or abbreviations and secondly because of the complexity of the solution space commonly present in many tasks that AI agents solve. One way of solving this is through a paradigm called LLM-as-judge.

LLM-as-judge involves, quite simply, prompting an LLM to grade the output or reasoning of the LLM agent. The LLM judge gets fed the trajectory of the LLM agent from an attempt at a task and outputs a grade, which could be on any scale, e.g. pass/fail or 1-5. LLM-as-judge is a good framework when the judging needs to be flexible and robust to subtle variations in the output of the agent [39]. However, since the sampling from LLMs is non-deterministic they introduce some amount of randomness in their grading which needs to be kept in mind during evaluation.

Besides being non-deterministic, LLM judges require alignment with domain experts so that the judgment they pass is as close to what a human would judge as possible. This is usually done by having a domain expert look at the judgments and then changing the prompt to steer the behavior of the LLM judge to something which aligns with the domain expert's judgment.

Further, recent work has shown that LLM judges can exhibit a phenomenon referred to as *same-family bias*, where outputs generated by LLMs from the same LLM family as the judge receive higher scores [42]. This bias has been observed even when controlling for differences in response quality. As a result, the choice of judge may influence the outcome of evaluations involving LLMs from different families and therefore needs to be considered when designing LLM-as-judge evaluation pipelines.

4

Context Adaptation

To improve the performance of LLM agents one can either improve the system around the LLM or the LLM itself. If one wants to improve the LLM, this can be done either through parameter updates or through something called context adaptation. Context adaptation involves adapting the prompts and context of the LLM and through this influencing the model behavior. Recent work has shown that this can be an effective way of improving LLM performance [15], [16].

This chapter details some of the more influential context adaptation methods. Starting from more simple paradigms like ICL and ending in state-of-the-art frameworks like ACE [15], [43].

4.1 In-Context Learning

Learning at inference time, i.e. after the LLM's parameters have been trained, has been studied as a mechanism for improving LLMs adaptability. Brown et al. introduced the term in-context learning (ICL) to describe how a pretrained LLM can perform a specific task by conditioning on information provided in its context [44]. In this setting, the model is given natural language instructions and example solutions, which it uses to infer the underlying task pattern and generate an appropriate response.

ICL is commonly categorized into three settings: zero-shot, one-shot, and few-shot learning. These differ in the number of examples provided to the model at inference time. A shot refers to an input-output example that demonstrates how the tasks should be solved, as illustrated in Figure 4.1.

Providing examples in the prompt improves performance because the examples implicitly specify the structure of the task and the mapping between inputs and outputs. Instead of relying only on knowledge stored in the LLM parameters which is acquired during training, the model can adapt its behavior to the current task by identifying patterns in the examples. This makes it possible for the model generalize to previously unseen inputs that follow a similar structure, even without updating the model's parameters.

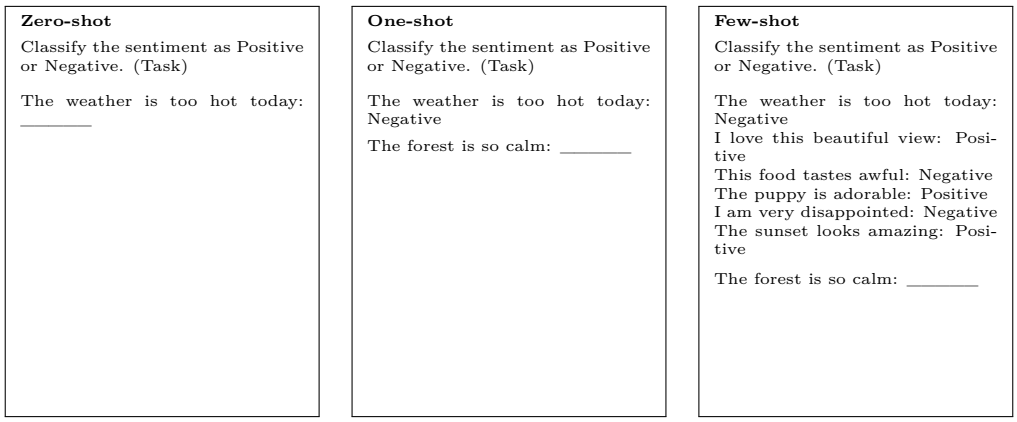

Figure 4.1: Examples of zero-shot, one-shot, and few-shot in-context learning.

Brown et al. [43] showed that few-shot learning improves performance compared to zero-shot learning on a range of tasks, including modeling long-range text dependencies, selecting appropriate endings for five-sentence stories and answering questions requiring broad factual knowledge. Their results further show that in-context learning performance scales with model size, with larger models achieving better results. Across the evaluated task, one-shot learning consistently outperformed zero-shot learning, while few-shot learning achieved the best performance.

As LLM context windows have increased, more examples can be included within a single prompt. Agarwal et al. [17] explored this extended context setting and introduced the concept of *many-shot learning*, i.e. introducing even more examples, showing further performance improvements over few-shot learning.

4.2 Chain-of-Thought Prompting

Chain-of-Thought prompting (CoT) was introduced by Wei et al. [45] to improve performance on tasks requiring multi-step reasoning, such as mathematical problems where intermediate calculations must be performed before arriving at the final answer. The idea builds on ICL by including examples that demonstrate not only the correct answer, but also the intermediate steps used to arrive at the answer.

The goal of CoT is to encourage the LLM to generate a *chain-of-thought*. One way to do this is by constructing ICL examples with intermediate reasoning steps [45], see Figure 4.2. The format consists of an input task, *chain-of-thought* and the output. The key step is the *chain-of-thought* description. It contains a sequence of intermediate natural language reasoning steps connecting the input task to the output. By conditioning its output on these types of examples, the LLM learns to generate similar intermediate steps when solving new tasks. This encourages the model to decompose complex problems into smaller and more manageable subproblems.

Since CoT is implemented entirely through prompting, it can be applied to pretrained LLMs without updating the LLM's parameters. The generated reasoning

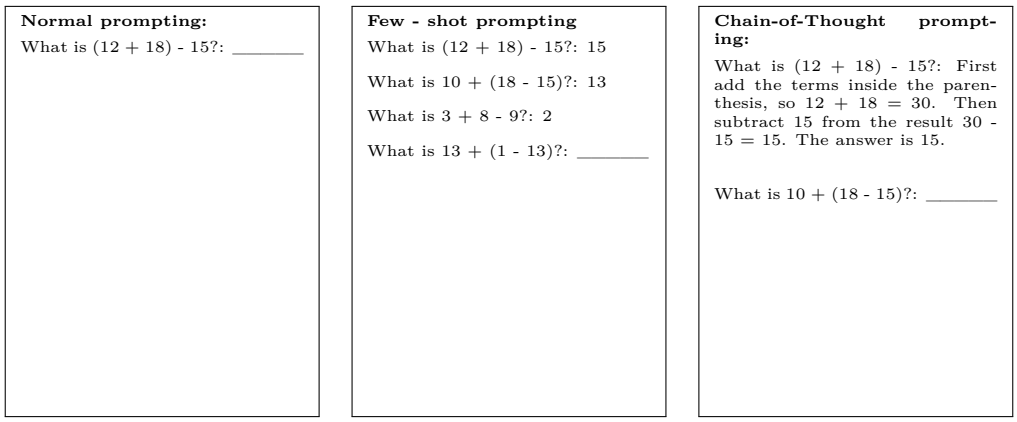

Figure 4.2: Illustration of normal, few-shot, and chain-of-thought prompting using a simple arithmetic task

steps also provides an interpretable view of the model's reasoning process, which can help identify where errors occur [45]. Furthermore, generating intermediate steps effectively increases the amount of computation allocated to complex tasks during inference, since the model generates more tokens.

By providing a few chain-of-thought examples, sufficiently large language models can learn to mimic these reasoning patterns and apply them to new tasks. Wei et al. [45] showed that chain-of-thought prompting leads to substantial performance improvements on several benchmarks. The improvements are most noticeable for more complex tasks that require intermediate reasoning steps, while simpler tasks show smaller gains. The effectiveness of CoT also increases with model scale, and is primarily observed in larger language models, where reasoning capabilities emerge more reliably.

Subsequent work has shown that CoT reasoning can also be used in a zero-shot setting without using any few-shot examples. The method is called zero-shot chain-of-thought and was introduced by Kojima et al. [46]. In this setting, a simple instruction such as "Let's think step by step" is appended to the prompt, encouraging the model to generate intermediate reasoning steps before producing an answer. The final answer is extracted from the generated output using a simple answer prompt or parsing step.

4.3 ReAct

Yao et al. [19] introduced a paradigm that aims to combine chain-of-thought reasoning with actions. While chain-of-thought prompting enables multi-step reasoning, it is limited to the knowledge encoded in the model parameters and cannot incorporate new information during inference. This can lead to incorrect reasoning steps and hallucinations, i.e. the LLM claiming something that is factually incorrect.

Reasoning and Acting (ReAct) addresses limitation by allowing the LLM to interact

with an external environment during reasoning, meaning LLM generation is interleaved with environment interaction. This way, the LLM both generates reasoning steps and actions, where the reasoning steps operate over the current context. The actions retrieve external information or perform operations that populate the context, allowing subsequent reasoning steps to make use of updated information, as illustrated in Figure 4.3. A prime example of these types of actions are tool calls.

ReAct System:

Input question: What is the population of the capital of Sweden?

Reasoning: I first need to identify the capital of Sweden.

Action: capital_search(sweden)

Action response: Stockholm is the capital of Sweden.

Reasoning: Now I should find the population of Stockholm.

Action: population_search(stockholm)

Action response: Stockholm has a population of about 1 million in the city.

Reasoning: I now have the answer.

Action: Finish(Approximately 1 million)

Figure 4.3: Example of the ReAct framework showing interleaved reasoning, tool use (action), and environment feedback (Action response).

By introducing both reasoning and acting, the LLM is extended into an agent capable of combining internal knowledge with external information sources. This allows the model to produce more grounded and accurate outputs by dynamically retrieving relevant information or performing external actions when needed. This interleaving of reasoning and acting is inspired by the observation that humans often exhibit this behavior when solving complex tasks [19], such as thinking through a problem while simultaneously consulting external information.

By interleaving reasoning and acting, the model can combine internal knowledge with external information, leading to more grounded and accurate outputs and actions. Empirically, ReAct has been shown to outperform methods based purely on reinforcement learning on several benchmark tasks [19].

4.4 Agentic Context Engineering

Agentic context engineering (ACE) is a prompt optimization framework introduced by Zhang et al. [15]. Prompt optimization is a paradigm in which instead of treating the context as a fixed instruction, you treat it as something which is adaptable and can be learned [47]. Prompt optimization frameworks learn the best prompt for the LLM by optimizing for some desired performance metric, e.g. pass@1.

In ACE, the context is treated as an evolving playbook of bullets consisting of reusable strategies, domain knowledge and lessons from past task attempts, see Figure 4.4. The bullets in the playbook are added, modified and replaced during the optimization. ACE enables language models to improve performance by iteratively refining the information contained within the context, i.e. playbook.

ACE Playbook:

STRATEGIES & INSIGHTS
[str-00001] helpful=5 harmful=0 :: Always verify data types before processing
[str-00002] helpful=3 harmful=1 :: Consider edge cases in financial data

FORMULAS & CALCULATIONS
[cal-00003] helpful=8 harmful=0 :: NPV = (Cash Flow / $(1+r)^t$)

COMMON MISTAKES TO AVOID
[mis-00004] helpful=6 harmful=0 :: Don't forget timezone conversions

Figure 4.4: Example of a small playbook context in ACE [48].

The motivation behind ACE stems from limitations observed in earlier context adaptation and prompt optimization methods, such as GEPA [16]. Prior approaches often relied on repeatedly rewriting or compressing prompts into concise summaries, which introduced what Zhang et al. [15] describe as *brevity bias*, where important domain-specific information and reasoning strategies are gradually removed in favor of shorter contexts. In iterative settings, these methods also suffered from *context collapse*, where accumulated knowledge deteriorates over time as prompts are repeatedly regenerated, leading to unstable behavior and loss of previously learned information. ACE addresses these issues by representing context as the structured and evolving playbook presented above. This allows ACE to preserve detailed knowledge, maintain long-term consistency and continuously improve agent behavior.

The optimization in ACE works by splitting the process into three distinct LLM roles: a generator, a reflector and a curator [15]. The generator interacts with the environment to solve tasks, typically using ReAct, producing both final outputs and intermediate reasoning trajectories. These trajectories are then analyzed by the reflector, which identifies errors, extracts key insights and diagnoses failures by comparing the generated output to expected outcomes and environmental feedback.

Importantly, ACE does not require the reflector to have access to ground truth labels, instead it can rely solely on execution feedback, e.g. task success or failure. The reflector is also tasked with tagging the bullets as helpful or harmful depending on their impact on the generator, as can be seen in Figure 4.4. Finally, the curator turns these reflections into incremental updates to the playbook, modifying or extending its contents with structured, reusable knowledge. ACE performs localized updates, i.e. considering each bullet on their own, rather than rewriting the entire context because this leads to more stable improvements while avoiding issues such as context collapse or loss of previously learned information. Bullets that accumulate a large number of harmful tags can be merged, modified, or removed from the playbook in order to prevent low-quality or misleading information from degrading performance over time.

ACE provides an alternative to traditional learning methods like parameter fine-tuning. Like other context adaptation methods it is particularly suited for settings where model weights cannot be modified. By operating entirely at the application layer, ACE is works well for closed-source LLMs and domain-specific agent systems, where it enables continuous improvement through interaction and reflection rather than retraining.

5

Methods

This chapter describes the methodology behind developing the prompt optimization framework for improving LLM agents in Recorded Future. The first sections below outline the data procurement, processing and preparation. Next, the structure and design of the optimization algorithm is described. Lastly, the design of the evaluation pipeline is reported.

5.1 Data Processing and Preparation

In order to apply the optimization framework, data is required for both training and evaluation. The framework was applied to two datasets: the *search* dataset and the *capture-the-flag* (CTF) dataset.

The search dataset consists of tasks that primarily require effective use of the TQL, the query language used to search in Recorded Future's data, and well-structured search strategies within this data. In contrast, the CTF dataset requires a broader set of capabilities, including general reasoning, multi-step problem solving, and the ability to integrate information from multiple sources. As a result, the two datasets place emphasis on different types of knowledge and capabilities.

5.1.1 Search Dataset

The search dataset consists of questions and expected answers related to recent news and cybersecurity. The queries in this dataset require the agent to construct syntactically correct TQL queries, specifying the correct date intervals, source categories and entity ids. To procure this data, news articles were collected and LLMs were applied to generate questions and answers.

5.1.1.1 Data Procurement

The first step of the data processing pipeline involved collecting articles from publicly available online news sources. The news article content was accessed programmatically, allowing relevant information to be extracted and analyzed automatically. First, links to available articles were collected from the target websites and stored

in a list. Each article was then accessed and processed individually. If an article matched the following constraints, its text was stored for question curation:

- The article contained one or more of the keywords seen in Table 5.1.
- The article was published in the last 60 days.
- The article had more than 20 words.

Table 5.1: Keywords, at least one needs to be in an article for it to be used to generate a data point.

Keywords	
war	conflict
military	missile
nuclear	cyber
hack	cyberattack
ransomware	malware
phishing	vulnerability
exploit	zero-day
threat	intelligence
breach	sanction
defense	

In total 130 articles were collected. The parameters were chosen to ensure that the articles were recent enough for the information to not be included in the LLM pretraining data and to ensure that articles were relevant to the threat intelligence domain.

5.1.1.2 Question Curation

After the data was collected, questions and answers were created. This was done by tasking an LLM with writing two high quality question and answer pairs for each collected article.

The instructions provided to the LLM included guidelines for how the questions should be generated. These guidelines focused on the structure and phrasing of the questions. For example, each question had to be answerable based on the information in the article while not being too tightly coupled to the specific article itself. This means that the question should also be answerable by reading one or more similar articles on the same topic. Using this approach, questions that are related to a single unique detail, opinion or other artifact that is present as a result of the specific writer or publication is avoided. To further improve qualitative performance of the questions, in-context learning was employed. This ICL features example questions representing how a user might structure its question to the system. This was added to better align the questions with real world questions.

To ensure variation in the generated questions, a set of predefined question categories was used, where each category provides guidance on how questions should be formulated. The different categories are listed in Table 5.2. The categories were designed to capture different types of information needs, such as causal relationships, affected entities, or temporal changes. During question generation, these categories were systematically applied to produce a balanced distribution, ensuring that no single question type was overrepresented. This approach enabled the generation of varied and comprehensive questions, improving both coverage and analytical depth.

Table 5.2: Question categories used for generating diverse questions

Category ID	Description
multi_part_fact_pair	Links two related facts (e.g., actor + action)
cause_and_impact	Explores causes and resulting consequences
scope_or_affected_targets	Identifies what was affected or at risk
timeline_or_change	Focuses on sequence or changes over time
comparison_or_distinction	Highlights differences between related items
entity_resolution	Requires identifying the correct entity
numeric_or_count_when_stable	Uses numerical data when stable

In total, 260 question and answer pairs are generated. These are then divided up into a training dataset with 200 questions, and an evaluation dataset with 60 questions.

5.1.2 Capture-the-flag dataset

The capture-the-flag (CTF) dataset was provided by Recorded Future. This dataset was smaller, consisting of 89 hard question regarding typical analyst tasks. These questions are harder since they require multi-step searching, comparisons, relationship extractions and other precise investigating capabilities. The 89 questions was split into 69 training questions and 20 evaluation questions.

5.2 Skills

One of the differences between a standard ACE implementation as presented in [15] and the implementation presented in this thesis is the usage of agent skills as a playbook. The skill that is optimized is called searching-recorded-future, and will be referred to as the search skill from now on.

The starting point during optimization for the search skill contained very little information, only a few examples as exemplified in Figure 5.1. The overall structure of it is very similar to what was used by Zhang et al. in [15].

5.2.1 Hand Written Skill

A skill that was hand written by Recorded Future was also used but only as a baseline to compare performance against. The exact content of the skill is confidential

```
---
name: searching-recorded-future
description: Minimal instructions for searching
   Recorded Future references and entities.
version: 0.0.1
---
# Searching Recorded Future
This skill details how to search Recorded Future

# STRATEGIES AND HARD RULES
## Section tag
    Section information
# TQLs TO USE FOR SPECIFIC INFORMATION

# USEFUL TQL SNIPPETS AND TEMPLATES
## TQL syntax example
    "Example with concrete TQL example"

# COMMON MISTAKES AND CORRECT STRATEGIES

# TROUBLESHOOTING AND PITFALLS


# OTHERS
```

Figure 5.1: The empty search skill.

information but Figure 5.2 contains an example of the general structure. The hand written skill contains information about how to access and query Recorded Future's data. Besides this, it also contains links to other files which the agent can access. These linked files contain more specialized information on how the query language and environment works.

```
---
name: searching-skill
description: Description of how to search Recorded
   Future as well as reference guides for Query
   language
    version: 0.0.1
---

# Searching Skill
This skill details how to search in Recorded Future

## Section 1
This is a section of information

## Section 2
‘‘‘
This is a section with code
‘‘‘

## Section 3
- This is
- a section with
- a bullet list

## Query Language Reference
- Linked file 1
- Linked file 2
- ...
```

Figure 5.2: Outline of the search skill provided by Recorded Future.

5.3 Optimization Algorithm Components

The implemented optimization algorithm is built on the methodology presented by Zhang et al. [15]. It is built on three core LLM components, the generator, reflector and curator. This section accounts for the implementation details of the algorithm.

5.3.1 Generator

The first part of the optimization algorithm is the generator. This agent is responsible for producing the solution trajectories and responses that either pass or fail a question, and it is the component the optimization algorithm seeks to improve. After training, the generator becomes the agent that end users interact with directly. Consequently, it has access to the same tools and operates under the same system prompt as the agent used in Recorded Future's system.

Two different LLM agents were used as generators. The "search agent" was applied to the search dataset, while the "CTF agent" was applied to the CTF dataset. The use of separate agents reflects the differing requirements of the two datasets. What differentiates them from each other are the tools. The search agent has a small amount of tools that are focused on searching with Recorded Future's query language TQL while the CTF agent has these tools as well as others for functionality such as fetching extensive metadata on specific entities. An example of some of the tools they have available is visible in Table 5.3.

Table 5.3: An abstract outline of the tools available to the generator.

Tool Name	**Tool Description**
Watchlists	Get information about what categories of information the user has explicitly marked as important to them.
Query by name	Find information about an event by the name of the event.
Query by TQL	Find information about events using TQL.
Count with TQL	Returns the amount of results the TQL query would return if executed
Load skill	Reads a skill from the skill server.

The generator works using the ReAct methodology to answer an input query. On startup the agent receives the system prompt along with descriptions of what tools it has available and the content of the search skill. This, along with the question, is fed into the LLM which it then conditions its output on. The agent now takes turns reflecting on what information it needs and how to get it, and acting by calling tools to fetch the information. Once the LLM finishes its generation the solution is complete. All the steps from initial query to reflecting and acting to final output is persisted in a list, this list is called the trajectory. The trajectory is fed into an evaluation pipeline to be graded and to get environment feedback which will be discussed below.

To generate the trajectory the generator takes the following as inputs: the system prompt, the query it needs to answer, the tools it has available, the search skill as well as an optional reflection. The optional reflection is described in more detail below.

5.3.2 Environment Feedback

After the generator has generated its trajectory it is passed along for collection of environment feedback. The environmental feedback comes in the form of an evaluation of the proposed answer given by the generator. The proposed answer generated by the generator is fed as input into two LLM judges that evaluate the answer and its trajectory. One of the judges is a correctness judge and the other is a grounding judge.

Both of the judges presented below were tuned using manual prompt engineering to align them with a domain expert. This was done through conversations with the domain expert and demonstrations of the judges. Through this, feedback was collected and the judges behavior was adapted through changing wording in the prompts or providing ICL examples. Further, the standard deviation of the judgments was checked on both datasets and was verified to be very low.

5.3.2.1 Correctness Judge

The correctness judge is tasked with evaluating if the predicted answer is correct and the task has been solved. Each answer is paired with the ground truth correct answer in the dataset and this is used by the judge to answer if the predicted answer is aligned with the ground truth. The judge outputs a pass or fail score and, for transparency, the judge also provides a rationale to support its decision.

The correctness judge differs somewhat between the two datasets. For the search dataset the correctness judge is quite lenient, giving a pass if the agent is able to find a source that refers to the same event that the question regards and if the agent answers the question, see Appendix B.1 for the exact prompt. The CTF dataset on the other hand has a much more strict judge which is prompted to verify that the exact keyword present in the dataset is also produced by the agent. More details on this can be seen in Appendix B.2.

5.3.2.2 Grounding Judge

The grounding judge is responsible for looking through the predicted answer and the generator's trajectory to determine if the predicted answer is aligned with the information it has retrieved from the tool calls. The grounding judge outputs a pass or fail score depending on whether the predicted answer is hallucinated by the generator or if the information is grounded in the retrieved information. The same grounding judge is used for both datasets.

5.3.3 Reflector

After collecting the environment feedback from the LLM judges, the trajectory and environment feedback are passed along to the reflector. The reflector is an LLM that is tasked with creating reflections based on the trajectory of the generator and environment feedback. The reflections that are produced are focused on the five key topics presented in Table 5.4.

Table 5.4: Reflection topics the reflector is tasked to reflect about.

Topic	Information
Reflection	The reflectors chain-of-thought/reasoning/thinking process, detailed analysis and calculations
Error identification	What specifically went wrong in the generators the reasoning?
Root cause analysis	Why did this error occur? What concept was misunderstood?
Correct approach	What should the model have done instead?
Key insight	What strategy, formula, or principle should be remembered to avoid this error?

Additional to these reflection topics, the reflector is also tasked with classifying some of the skill's sections as helpful or harmful. This is included to provide an additional signal the curator for how the sections are contributing to the behavior of the generator.

To generate effective reflections, the reflector consumes a rich set of inputs. These include the original questions the generator attempted to solve, as well as the generator's full trajectory, including tool calls and their responses, intermediate reasoning steps, and the final output. In addition, the reflector is provided with the current state of the skill, environmental signals, the predicted answer, and the ground truth answer.

To further enhance the quality of the reflections produced by the reflector, additional configurations were explored.

5.3.3.1 Reflector with Documentation

The first additional reflector configuration, referred to as the *reflector with documentation*, augments the reflector by injecting a subset of TQL documentation into its system prompt. The purpose of this modification is to enable the generation of more accurate and grounded reflections. In this configuration, the reflector is additionally tasked with guiding the agent in the use of TQL and is instructed to cross-reference query-related suggestions with the provided documentation to ensure syntactic correctness. This allows the reflector to produce more precise and actionable feedback, particularly for query construction and refinement.

5.3.3.2 ReAct Reflector

The second additional reflector variant was developed based on the ReAct paradigm. In this version, the reflector is implemented as a ReAct agent, meaning that it

is capable of both reasoning and interacting with the same tools available to the generator.

The motivation for this design is that active interaction with the environment may enable a deeper understanding of query strategies, tool usage, and the structure of the data. This capability allows the reflector to explore alternative query formulations, verify assumptions, and effectively fact-check the generator's behavior. Such functionality is expected to improve the quality and grounding of the reflections produced.

Through this interaction with the environment it was hypothesized that the reflector could try out slight variations in the generator's trajectory and through that analyze if the answer it arrived at was the correct answer, even without any ground truth or environment feedback from the correctness judge. Hence, a ReAct reflector configuration that was not provided with the ground truth or any environment signal from the correctness judge was also implemented. This configuration is inspired by the removal of ground truth from the reflector explained in Section 4.4 but is taken a step further with the complete removal of correctness feedback.

5.3.4 Curator

The final step in the chain is the curator. This is an LLM tasked with updating the search skill based on the reflections produced by the reflector. Similarly to the reflector, the curator gets the original question the generator attempted to solve and the current state of the skill and its associated files as input. Additionally to this the curator also gets the reflection produced by the reflector, a minimal changelog of what the curator has updated in previous iterations and a set of metadata related to the current training. The metadata consists of additional information that the curator should take into account when performing updates. Information such as total sections across the skill, the total amount of tokens the skill consists of and the remaining token budget for updates, how many sections of the skill are high performing, unused or problematic when the generator is attempting to perform the task. The high performing, unused and problematic fields are determined as seen in Algorithm 1, in accordance with [15].

Algorithm 1 Section Performance Classification

1: **if** `helpful` > 5 **and** `harmful` < 2 **then**
2: classify as high-performing
3: **end if**
4: **if** `harmful` $>$ `helpful` **then**
5: classify as problematic
6: **end if**
7: **if** `helpful` $+$ `harmful` $= 0$ **then**
8: classify as unused
9: **end if**

Based on this information, the curator is responsible for generating delta updates to the skill. These updates represent incremental modifications rather than full rewrites, enabling controlled and efficient evolution of the skill over time. The delta updates are returned by the curator in the structured format shown in Figure 5.3. The curator operates through a limited set of predefined actions.

```
{
  "reasoning": "brief rationale",
  "operations": [
    {
      "type": "ADD | UPDATE",
      "skill": "searching-recorded-future",
      "parent_section": "master section name",
      "section": "section name",
      "content": "short reusable bullet-tag content"
    }
  ]
}
```

Figure 5.3: Structured output format used by the curator to express reasoning and skill update operations.

The first action, ADD, introduces a new section containing previously absent content, allowing the skill to expand its coverage or incorporate newly identified knowledge. The second action, UPDATE, modifies an existing section by refining or replacing its content, typically in response to identified deficiencies or performance issues.

5.3.4.1 Skill Updates

By constraining the curator to only ADD and UPDATE operations, the resulting delta updates remain small and focused. This design aligns with the ACE framework and promotes localization, fine-grained retrieval, and incremental adaptation [15].

Once the curator has generated the updates, the current version of the skill is modified accordingly. During this process, the parent_section field, as shown in Figure 5.3, is used to match the appropriate location in the existing structure, ensuring that new or updated sections are correctly placed within the skill.

The sections generated by the curator are persisted in the skill. However, sections that accumulate more than ten harmful tags from the reflector are automatically removed during optimization. While Zhang et al. [15] do not explicitly describe a pruning strategy of this kind in ACE, this mechanism was introduced in this work to remove consistently underperforming sections from the skill. The underlying hypothesis is that pruning harmful sections reduces the risk of degenerative performance and prevents low-quality information from accumulating during optimization.

5.4 Training Loop

The training loop consists of orchestrating the three agents, generator, reflector and curator. All LLMs in the optimization are OpenAI's GPT-5.4-mini [49]. The implementation can be seen in Algorithm 2, where $\mathcal{D}$ is the dataset.

Algorithm 2 ACE Training Loop with Iterative Refinement

```
for query, ground_truth ∈ 𝒟 do
    trajectory ← GENERATOR(query)
    score ← LLMJUDGE(trajectory, ground_truth)
    if score is fail then
        for i = 1 to N refinement steps do
            reflection ← REFLECTOR(query, trajectory, ground_truth, score)
            trajectory ← GENERATOR(query, reflection)
            score ← LLMJUDGE(trajectory, ground_truth)
            if score is pass then
                break
            end if
        end for
    else
        reflection ← REFLECTOR(query, trajectory, ground_truth, score)
    end if
    deltas ← CURATOR(query, reflection)
    apply_skill_updates(deltas)
end for
```

Step-by-step the algorithm works by taking a data point from the dataset and first feeding the query to the generator. The generator attempts to answer the query by looking in the Recorded Future data through the use of different tools. When the trajectory has been generated it is fed to the LLM judges to give environment feedback. The trajectory along with the environment feedback is then fed to the reflector.

The reflector assesses the trajectory of the generator using the environment feedback and the ground truth output to analyze where the generator went wrong and right. If the generator failed the task the reflection is optionally fed back to be used by the generator for another attempt at the task. This generator-reflector loop is repeated a maximum of n times but ended if the generator gets a pass from the environment feedback. Each time the reflector reflects it also tags the different section of the search skill as either helpful, harmful or neutral. These tags are persisted and used by the generator and curator. After these iterations the reflection by the reflector is passed along to the curator.

The curator takes the reflection and search skill along with its section tags and produces operations of one of two types, add or update, as described above. These operations are then applied to the search skill and training on one data point is complete and the loop moves on to the next data point.

5.5 Evaluation

An evaluation pipeline was implemented in order to track improvement and validate the implemented optimization algorithm. The two datasets were split into held-out test splits as described above.

5.5.1 Metrics

The main metrics used to evaluate the LLM agents were, task success accuracy as concluded by the correctness LLM-judge measured in pass@k and pass^k for $k \in \{1, 2, 3\}$. The LLM agents were also evaluated using the grounding judge with the same pass@k and pass^k setup.

Other metrics that were used to evaluate the training efficiency, but not optimized for, were token usage and latency. This gave an indication on how many tokens were used for the optimization which affects both cost and the environment. Further the latency highlighted how much time the optimization took to run.

5.5.2 Baseline & Algorithm evaluation

The baselines and the optimized LLM agents were evaluated on the exact same pipeline, using the same metrics and the same data to ensure fairness. The agents were fed the input query from the validation dataset. They produced their trajectory which was then evaluated by an LLM-judge for correctness and using the other metrics described above.

5.5.3 Hypothesis Testing for Agent Comparison

To test whether an optimized agent is better than a baseline agent with statistical significance, z-tests were conducted. The setup of these z-tests is described here.

Each agent is evaluated on a validation dataset consisting of N questions where each question is attempted K times. If one lets $X_{i,j}, Y_{i,j} \in \{0, 1\}$ be the outcome of attempt $j \in [1, K]$ of question $i \in [1, N]$ for the baseline and optimized agent respectively. One can define estimators for the pass@1 (pass rate) for each question i as:

$$\bar{X}_i = \frac{1}{K} \sum_{j=1}^{K} X_{i,j}, \ \bar{Y}_i = \frac{1}{K} \sum_{j=1}^{K} Y_{i,j}$$

Using this, one defines the difference between the optimized agent's performance and the baseline agent's performance for question i as:

$$\Delta_i = \bar{Y}_i - \bar{X}_i$$

With the sample mean, $\bar{\Delta}$, defined as:

$$\bar{\Delta} = \frac{1}{N}\sum_{i=1}^{N}\Delta_i$$

With the above definitions, one would like to perform the following one-sided hypothesis test:

$$H_0 : \mathbb{E}[\bar{\Delta}] = 0,\ H_1 : \mathbb{E}[\bar{\Delta}] > 0$$

To conduct this test one needs to compute the variance $\mathrm{Var}(\bar{\Delta})$. To do this, first let $p_i = P(X_{i,j} = 1)$. Under H_0, it then follows that $P(Y_{i,j} = 1) = p_i$ and also that $\mathrm{Var}(\Delta_i) = \frac{2}{K}p_i(1-p_i)$. Hence, $\mathrm{Var}(\bar{\Delta}) = \frac{2}{KN^2}\sum_{i=1}^{N} p_i(1-p_i)$, where p_i can be substituted for $\hat{p}_i$, meaning:

$$\hat{p}_i = \frac{1}{2K}(\{\text{amount of observed } X_{i,j} = 1\} + \{\text{amount of observed } Y_{i,j} = 1\})$$

Now construct the z-test with test statistic Z:

$$Z = \frac{\bar{\Delta}}{\sqrt{\mathrm{Var}(\bar{\Delta})}}$$

with $Z \sim N(0,1)$ under the null-hypothesis which can be rejected with $p = \alpha$ if $Z > z_{1-\alpha}$.

5.6 Objective

The optimization is applied to two datasets starting from the empty search skill described above. The different ACE configurations with different reflector variants is compared against the two baselines and against each other. The results give indication on how well ACE can be applied in industry as well as how different implementations of the reflector affect the performance.

The objective of this thesis is primarily to evaluate the performance of the optimization framework on the search dataset. In addition to the search dataset, the CTF dataset is included as an auxiliary benchmark to examine how the methods generalize to a different and more difficult, yet related problem setting.

6

Results

This chapter presents the results obtained from applying the proposed optimization framework to the two datasets described in Chapter 5. The results are first presented for the search dataset, which serves as the primary evaluation setting, followed by the capture-the-flag dataset to assess generalization to more complex tasks. The amount of refinement steps was set to one across all settings and datasets.

6.1 Search Dataset

The search dataset is the primary evaluation setting for the optimization framework. As described in Chapter 5, this dataset consists of queries that require retrieving information about different events in Recorded Future's data sources. These queries were designed to reflect realistic user interactions with the search agent. To evaluate the effectiveness of ACE in this setting, four experimental setups were considered. First, a standard ACE setup with general-purpose prompts similar to those presented by Zhang et al. in [15]. This configuration serves as a reference for understanding how well ACE performs without domain adaptation of the framework. Second, an ACE setup where the reflector has access to TQL documentation. This configuration was adapted to the Recorded Future environment and the specific task, where both the reflector and curator were explicitly guided to generate and refine documentation related to TQL. Third, a more advanced configuration where the reflector was extended with ReAct capabilities. This allowed it to interact with the same tools as the generator and explore and verify its reflections. Finally, a reflector with ReAct capabilities but that was not given any ground truth or correctness environment feedback.

All configurations were trained over a single epoch of the 200 training queries. The resulting agents were tested on a held out test set of 60 queries using the metrics described in Section 5.5

6.1.1 Training

Here the computational aspects of the ACE configurations are presented in terms of average generation time and average tokens per generation. These metrics provide

insight into the efficiency and scalability of the optimization framework, complementing the performance evaluation presented later.

Table 6.1 summarizes the average time and token usage per training iteration across the evaluated configurations. A training iteration includes all steps where the generator attempts to solve a query, involving multiple reasoning steps, tool calls, and intermediate outputs and the reflector and curator results. Since the ACE framework relies on iterative refinement through generator–reflector-curator interactions, the cost of each generation is an important factor influencing overall training efficiency.

Table 6.1: Average generation time and token usage across different settings.

Reflector Configuration	**Mean**		**Median**	
	Time (s)	**# Tokens**	**Time (s)**	**# Tokens**
Standard	28.4	117 000	26.3	105 000
With Docs	40.0	162 000	37.8	127 000
ReAct	61.4	186 000	56.6	189 000
ReAct (No GT)	56.3	192 000	54.7	182 000

As one sees in Table 6.1 each training iteration requires a substantial amount of tokens. However, one has to keep in mind that a strong majority of these tokens are input tokens from tool descriptions, responses and system prompts. On average across all settings the amount of output tokens is only around 2000 per training iteration.

The results show that more advanced configurations tend to incur more computational costs, especially those using ReAct-enabled reflectors. This is expected since allowing the reflector to interact with tools introduces additional reasoning steps, tool calls and intermediate results, which increases both the time and amount of tokens it takes to execute an iteration of training. Similarly, giving the reflector the TQL documentation expands the system prompt and context, which contributes to higher token usage and time per iteration. This is illustrated in Figure 6.1, where one can see the individual components average time for the generation.

6.1.2 Evaluation

The evaluation on the search dataset is conducted using the metrics described in Section 5.5. The focus lies on comparing the trained agent with optimized skills to the two baselines with empty and handwritten skills, in terms of pass@k and pass^k for correctness and grounding.

Table 6.2 shows the performance across agents using pass@k and pass^k for $k \in \{1, 2, 3\}$. The trained configurations consistently outperform the baselines across all values of k. This performance increase is consistent even when comparing to

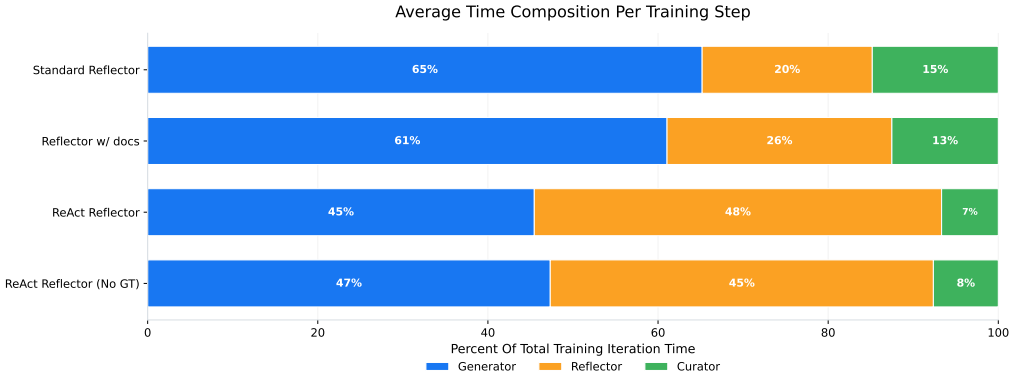

Figure 6.1: Average training iteration time composition across training on search dataset. Each horizontal bar is normalized to 100% of the total iteration time and shows the relative contribution of the generator, reflector, and curator components.

the handcrafted baseline which shows that the methods presented in this paper can outperform manual context engineering by domain experts.

Table 6.2: Performance comparison across agents using pass@k (correctness) and pass^k for $k = 1, 2, 3$. Changes are relative to the Empty Skill baseline (percentage points).

Setting	**pass@k**			**pass^k**		
	@1	@2	@3	^1	^2	^3
Empty Skill	73.7	88.3	93.4	73.7	59.1	49.6
Handcrafted Skill	79.0$^{+5.3}$	89.6$^{+1.3}$	93.2$^{-0.2}$	79.0$^{+5.3}$	68.4$^{+9.3}$	61.4$^{+11.8}$
Reflector Configurations						
Standard	82.3$^{+8.6}$	92.7$^{+4.4}$	96.0$^{+2.6}$	82.3$^{+8.6}$	72.0$^{+12.9}$	65.0$^{+15.4}$
With Docs	84.0$^{+10.3}$	92.3$^{+4.0}$	95.3$^{+1.9}$	84.0$^{+10.3}$	75.7$^{+16.6}$	70.5$^{+20.9}$
ReAct	86.3$^{+12.6}$	94.9$^{+6.6}$	97.2$^{+3.8}$	86.3$^{+12.6}$	77.8$^{+18.7}$	71.6$^{+22.0}$
ReAct (No GT)	**88.8**$^{+15.1}$	**97.4**$^{+9.1}$	**99.3**$^{+5.9}$	**88.8**$^{+15.1}$	**80.2**$^{+21.1}$	**73.5**$^{+23.9}$

An interesting observation is that the configuration trained without access to ground truth signals (No GT) is still able to achieve competitive performance. Despite not receiving explicit correctness feedback during training, this configuration performs on par with configurations that do make use of such signals.

Another interesting observation is that the pass@1 for the reflector with documentation is greater than the pass@1 for the standard reflector, however this is not the case for pass@3. Suggesting that the agent optimized with the standard reflector could potentially meet the performance of the agent optimized with the reflector with documentation when given more attempts. This is something that is touched upon in greater detail in the discussion below.

In Table 6.3, the results are presented in terms of correctness and grounding using

pass@1. The trained agents show improvements in correctness compared to the baselines, while still maintaining similar levels or improving upon the grounding. However, no grounding score is greater than the correctness score which is something that is discussed in the next chapter.

Table 6.3: Performance comparison across settings on correctness and grounding using pass@1. Changes are relative to the Empty Skill baseline (percentage points). Best results per column are bolded.

Setting	**Correctness (pass@1)**	**Grounding (pass@1)**
Empty Skill	73.7	60.7
Handcrafted Skill	79.0$^{+5.3}$	61.3$^{+0.6}$
Reflector Configurations		
Standard	82.3$^{+8.6}$	58.5$^{-2.2}$
With Docs	84.0$^{+10.3}$	62.2$^{+1.5}$
ReAct	86.3$^{+12.6}$	**74.5**$^{+13.8.}$
ReAct (No GT)	**88.8**$^{+15.1}$	72.0$^{+11.3}$

To further analyze the results, hypothesis testing is performed between the trained configurations and the baselines, as shown in Table 6.4. The reported test statistics and p values indicate that all of the observed improvements are statistically significant, i.e. null-hypothesis can be rejected at 95% confidence. The smallest p-value is achieved when comparing the ReAct reflector with the empty skill baseline and the largest p-value is observed when comparing the standard reflector with the handcrafted skill baseline.

Table 6.4: Comparison of trained settings against the Empty Skill and Handcrafted Skill baselines. The table reports the test statistic (z) and the one-sided p-value. Best results per column are bolded.

Reflector Configuration	**z-statistic**		**One-sided p-value**	
	vs Empty	**vs Handcrafted**	**vs Empty**	**vs Handcrafted**
Standard	4.211	1.782	1.27e-05	3.73e-02
With Docs	5.303	2.717	5.70e-08	3.29e-03
ReAct	6.421	4.010	6.77e-11	3.04e-05
ReAct (No GT)	**7.650**	**5.340**	**9.99e-15**	**4.75e-08**

Overall, the results show that the trained configurations improve performance on the search dataset compared to empty skill baseline and the handcrafted skill baseline in all cases. Furthermore, in Table 6.5 the reflector variations are compared against each other in hypothesis tests, highlighting how both ReAct reflector variations have significantly improved over the standard reflector.

Table 6.5: Pairwise p-values between configurations

		Comparison Configuration			
		Standard	With Docs	ReAct	ReAct (No GT)
Baseline	Standard	–	1.73e-01	1.18e-02	1.27e-04
	With Docs	8.27e-01	–	8.12e-02	2.26e-03
	ReAct	9.88e-01	9.19e-01	–	6.59e-02
	ReAct (No GT)	1.00e+00	9.98e-01	9.34e-01	–

6.2 Capture-the-flag Dataset

The capture-the-flag dataset (CTF dataset) is a more challenging setting compared to the search dataset. As described in Section 5.1.2, these tasks require multi-step reasoning, precise query formulation, and the ability to iteratively refine search strategies in order to arrive at a correct answer.

In contrast to the search dataset, which focuses on retrieving information related to some event queried by the user, the CTF dataset focuses on deeper investigation, complex reasoning and strict factual correctness. This makes the dataset suitable for evaluating the optimization framework in more demanding scenarios.

Like the experiments conducted on the search dataset, four configurations were evaluated. First, a setup with a standard reflector using general-purpose prompts was used as reference. Second, a configuration with domain-specific documentation provided to the reflector. Finally, a configuration with a ReAct-enabled reflector, with and without ground truth feedback.

All configurations were trained for one epoch over the 69 training questions and evaluated on the held-out test set of 20 questions.

6.2.1 Training

The computational characteristics of the ACE configurations on the CTF dataset follow similar trends to those observed for the search dataset, but with consistently longer generation times and token usage due to increased task complexity. As discussed in Section 5.1.2, these tasks require longer reasoning chains, iterative query refinement, and exploration of intermediate hypotheses before reaching a final answer.

Table 6.6 summarizes the average generation time and token usage per training iteration across the evaluated configurations. The need for more sequential tool interactions and extended reasoning traces leads to longer computation time per generation.

The computational overhead is particularly evident for the more advanced reflector configurations. ReAct-based reflectors, in particular, require substantially more time

Table 6.6: Average generation time and token usage across different settings.

Reflector Configuration	Mean		Median	
	Time (s)	# Tokens	Time (s)	# Tokens
Standard	69.7	89,319	61.1	81,306
With Docs	59.3	101,015	53.5	82,473
ReAct	123.0	175,024	118.5	154,579
ReAct (No GT)	114.5	162,070	103.7	146,767

due to additional reasoning steps and tool interactions during reflection, resulting in longer trajectories.

The general trend between the reflector configurations is similar to the search dataset, where the ReAct-enabled reflector an the additional TQL documentation contributes to higher token usage. This is also illustrated in Figure 6.2, which breaks down the average time spent across generator, reflector, and curator stages for each configuration.

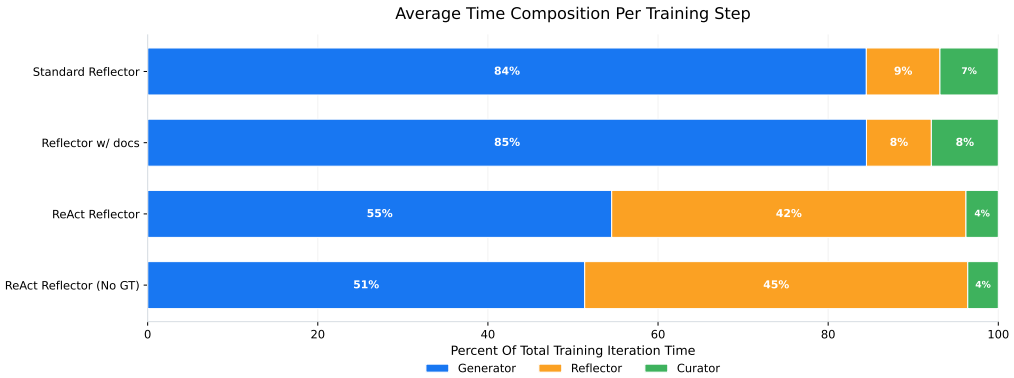

Figure 6.2: Average rollout iteration time composition across training on the CTF dataset. Each horizontal bar is normalized to 100% of the total iteration time and shows the relative contribution of the generator, reflector, and curator components.

Despite the increased computational time, the results indicate that the optimization framework remains feasible to train in more demanding environments, although the added reasoning capabilities of the ReAct-based reflectors come at a clear computational trade-off. The increased computational time is also expected, since the CTF environment provides the agent with access to more tools and requires more complex interactions than the search dataset.

6.2.2 Evaluation

The evaluation on the CTF dataset is conducted using the same metrics as for the search dataset, i.e. pass@k and pass^k for correctness and grounding. The results are presented in Tables 6.7 and 6.8.

Table 6.7: Performance comparison across agents using pass@k (correctness) and pass^k for $k = 1, 2, 3$. Changes are relative to the Empty Skill baseline (percentage points).

Setting	**pass@k**			**pass^k**		
	@1	@2	@3	^1	^2	^3
Empty Skill	32.0	44.8	51.4	32.0	19.2	13.1
Handcrafted Skill	34.0$^{+2.0}$	44.9$^{+0.1}$	49.3$^{-2.1}$	34.0$^{+2.0}$	23.1$^{+3.9}$	16.6$^{+3.5}$
Reflector Configurations						
Standard	37.2$^{+5.2}$	49.4$^{+4.6}$	55.9$^{+4.5}$	37.2$^{+5.2}$	24.6$^{+5.4}$	17.8$^{+4.9}$
With Docs	**41.8** $^{+9.8}$	**54.1**$^{+9.3}$	**59.8**$^{+8.4}$	**41.8**$^{+9.8}$	**29.4**$^{+10.2}$	**22.9**$^{+9.8}$
ReAct	38.0$^{+6.0}$	49.7$^{+4.9}$	55.8$^{+4.4}$	38.0$^{+6.0}$	26.3$^{+6.9}$	20.8$^{+7.7}$
ReAct (No GT)	38.5$^{+6.5}$	49.8$^{+5.0}$	55.7$^{+4.3}$	38.5$^{+6.5}$	27.2^{+8}	21.8$^{+8.7}$

Table 6.8: Performance comparison across settings on correctness and grounding using Pass@1. Changes are relative to the Empty Skill baseline (percentage points). Best results per column are bolded.

Setting	**Correctness (Pass@1)**	**Grounding (Pass@1)**
Empty Skill	32.0	56.8
Handcrafted Skill	34.0$^{+2.0}$	**62.3** $^{+5.5}$
Reflector Configurations		
Standard	37.2$^{+5.2}$	59.8$^{+3.0}$
With Docs	**41.8**$^{+9.2}$	56.2$^{-0.8}$
ReAct	38.0$^{+6.0}$	49.2 $^{-7.6}$
ReAct (No GT)	38.5$^{+6.5}$	60.5$^{+3.7}$

Compared to the search dataset, overall performance is lower across all configurations, reflecting the increased difficulty of the CTF tasks, which require more complex interaction with the environment and a larger set of tools. The empty skill baseline achieves a pass@1 of 32.0%, indicating a substantially more challenging setting for the agent.

The results in Table 6.7 show consistent improvements across all ACE configurations compared to the baselines, although the magnitude of gains varies between settings. In particular, configurations with stronger reflection mechanisms and additional contextual information tend to perform better across all values of k. The largest improvements are observed for the reflector with documentation configuration, which consistently achieves the highest scores across both pass@k and pass^k. Overall, the results indicate that richer reflective guidance contributes positively to performance in the CTF setting.

The grounding results are mixed, with some configurations improving over the baseline while others decrease, particularly for the ReAct-based reflector. This indicates a trade-off between correctness and grounding across different configurations.

Statistical testing, shown in Table 6.9 indicates that several of the observed improvements are statistically significant at the 95% confidence level, particularly for configurations incorporating stronger reflection mechanisms and richer contextual guidance. Overall, the results indicate that the optimization remains effective in

Table 6.9: Comparison of trained settings against the Empty Skill and Handcrafted Skill baselines. The table reports the test statistic (z) and the one-sided p-value. Best results per column are bolded.

Reflector Configuration	**z-statistic**		**One-sided p-value**	
	vs Empty	**vs Handcrafted**	**vs Empty**	**vs Handcrafted**
Standard	2.051	1.286	2.02e-02	9.92e-02
With Docs	**3.845**	**3.199**	**5.91e-05**	**6.90e-04**
ReAct	2.350	1.593	9.39e-03	5.56e-02
ReAct (No GT)	2.567	1.841	5.12e-03	3.28e-02

complex, multi-step tasks, but its success is more dependent on the strength of the reflector and the availability of informative feedback signals. Moreover, only the reflector with documentation is outperforming the standard reflector with statistical significance, i.e. the p value is less than 0.05. This can be seen in Table 6.10.

Table 6.10: Pairwise p-values between configurations

		Comparison Configuration			
		Standard	With Docs	ReAct	ReAct (No GT)
Baseline	Standard	–	3.55e-02	3.80e-01	3.03e-01
	With Docs	9.65e-01	–	9.35e-01	9.09e-01
	ReAct	6.20e-01	6.50e-02	–	4.17e-01
	ReAct (No GT)	6.97e-01	9.12e-02	5.83e-01	–

7

Discussion

This chapter interprets and contextualizes the results presented in Chapter 6. Instead of focusing only on the quantitative performance, this discussion presents the underlying causes for the observed behavior of the different configurations. First, the results across the two datasets are compared and broader insights are drawn regarding ACE in this niche domain environment. The chapter then concludes by highlighting limitations of the study and directions for future work.

7.1 Comparison and Insights

This work aimed to evaluate context engineering using ACE for LLM agents in niche, real-world domains. The results show clear and consistent improvements over the empty skill baseline. Further, the handcrafted skill baseline is improved upon in most cases. This shows that ACE can outperform contexts crafted by humans in these niche environments. However, besides these quantitative performance gains, several insights into why this is the case are highlighted here.

7.1.1 Correctness and Grounding

One quite consistent trend throughout the results is that the gains in correctness are larger and more steady as compared to the grounding. The main goal for this work is improving correctness but that is never explained to the reflector or curator, i.e. there is no part of the prompt that says something like "prioritize correctness over grounding." One potential reason then for this disparity is that the correctness metric is more closely related to the overall narrative in the reflector and curator prompts. Meaning that the prompts are framed around helping the generator solve tasks and correctness is more aligned with this objective than grounding. If grounding was more important, then one would have to highlight how a solution without grounding isn't a valid solution more clearly and through that encourage the reflector and curator to create updates to help with grounding.

Another thing to note is that it isn't necessarily expected for the grounding and correctness to be correlated. A poor agent which seldom finds answers to the queries can often output that it did not find anything and that is a grounded answer and hence the grounding score for that agent is high. Similarly, a good agent that often

finds the correct answers might occasionally output a hallucinated answer because it has been learned during training that it is good to always output an answer.

However, none of the above describe why the grounding pass@1 is consistently lower than correctness for the search dataset. Furthermore, it does not explain why the grounding appears to fluctuate around approximately 60% for almost all settings. Therefore, another possible, and perhaps more probable, explanation to why performance is not improving as consistently for the grounding judge is that the grounding judge is not performing as well at its task as one might have hoped. None of the judges were evaluated quantitatively to check how they perform and the majority of time was spent qualitatively verifying and tuning the correctness judges, since they were the primary aim of the optimization. Hence, one can be quite assured of the correctness judge performance but the grounding judge could possibly be quite poor at verifying grounding. This could then result in the grounding performance observed in the presented results. Another finding underpinning this insight is that, qualitatively, no correct answer from the generator was ever observed to be without grounding.

7.1.2 Reflector Variations

Another, perhaps more important, finding is that the performance of the optimization appears to be heavily influenced by the quality of the reflector's feedback. Here, "quality" refers to how informative, specific, and actionable the generated reflections are. The ReAct reflector and the reflector with documentation outperform the standard reflector because they are able to provide more specific error signals, reduce ambiguity in how failures should be corrected, and enable the reflector to verify reflections rather than merely speculate about possible improvements.

An important consideration is that the underlying language model has limited prior knowledge of the TQL language, the associated tools and Recorded Future's environment as a whole, since it is proprietary and not widely represented in public training data. As a result, the standard reflector often struggled to generate detailed and technically accurate reflections related to TQL syntax and query construction. The generated reflections therefore tended to remain relatively general, frequently suggesting high-level improvements without providing concrete examples or reliable guidance on how the search queries should be structured. On the other hand, by allowing the reflector to retrieve information from documentation or interact with tools, the generated reflections were grounded in valid TQL syntax and concrete query examples, rather than relying on the model to infer or approximate the syntax from prior knowledge alone.

The results indicate a trade-off between reflection precision and generality across the evaluated configurations. The more advanced variants, specifically the ReAct reflector and the reflector with documentation, produce precise feedback, which means more stable performance as is reflected in the higher pass@1 metrics. On the other hand, the standard reflector generates broader, more generalized reflections. While these general insights lead to lower initial performance at pass@1, they

introduce some hints at the correct path that allows the generator to occasionally discover the correct reasoning on the more difficult questions when given multiple evaluation attempts. In Table 6.2, the standard reflector performs worse than the documentation-enabled model at pass@1 but reverses this trend at pass@3. A similar case can be seen in Table 6.7 when comparing the standard reflector against the ReAct configuration, hinting that less specialized reflectors mean higher stochasticity but still remain capable of finding solutions as k increases.

Another thing to note is that the ReAct reflector does not perform as well on the CTF dataset as on the search dataset. One potential reason for this could be that the way the search dataset is setup, it benefits from broad exploration and flexibility in the queries. This could be because the events are much easier to find than the more specific items in the CTF dataset. This way the ReAct reflector is able to use its ReAct capabilities effectively in order to produce high quality reflections. However, the CTF dataset requires precise reasoning and queries in order to find the ground truth items. Because of this the ReAct reflector is more seldom able to find the correct answer and hence its interactions with the environment introduce more noise than help when generating the reflections.

An additional insight is that the ReAct reflector version without ground truth seem to consistently produce competitive performance as well as outperform the ReAct reflector with ground truth. This indicates that useful learning signals can come from just interacting with the environment and that ground truth supervision for ACE might not be strictly necessary if the feedback signals that the reflector is able to produce on its own are strong. This result also supports the argument for ACE-like frameworks in real-world industrial settings, where labeled data is scarce or expensive. This might seem like a counter-intuitive result however one potential reason for why this is the case is that the actual strongest learning signal in these datasets is not the ground truth answer but that it is in fact the trajectories. The trajectories contain missteps, tool call failures etc. and this appears to be enough of a learning signal to effectively enhance the context of the agent, at least in this setting.

7.1.3 Question Curation

Another thing that deserves some discussion is the question curation for the search dataset. The question curation was set up as a programmatic pipeline that takes news articles and feeds them to an LLM to generate a dataset. Creating questions using an LLM like this does come with some risks like hallucinating the correct answer or possibly creating the wrong types of questions. However, since the requirements on the search agent finding the exact correct ground truth answer was lenient and only finding information regarding the same event was required, smaller variations in the ground truth created by the LLM was acceptable. This meant that the LLM question curation was an effective way of creating data in this work. Furthermore, the questions were qualitatively validated by us to assure that the questions were of the expected format and type.

7.2 Limitations and Future Work

While the results show strong performance improvements, some limitations inhibit the conclusions and point toward directions for future work.

7.2.1 Training

Firstly, the optimization was performed over a single epoch without collection of any data that regards training progress. As a result, it is unclear whether performance had already converged before the epoch finished, or whether further improvement would have been possible with additional training. This introduces some uncertainty regarding the true upper bound on performance and whether the different configurations mainly differ in sample efficiency while ultimately converging toward similar results. However, during testing, additional epochs did not produce any noticeable improvement in performance, which is why the training was not repeated for longer durations. Further, since evaluation took a long time and cost money, per iteration metrics were infeasible to collect.

Another limitation regarding training is the limited data available for the CTF dataset. It requires more complex trajectories and more diverse interactions with tools than the search dataset, i.e. the amount of possible trajectories for the agent is larger. This tends to mean that more data is required in order to properly learn the best strategy for the agent to follow. Yet, the CTF dataset is smaller than the search dataset. It is therefore hypothesized that a larger and more diverse dataset on tasks similar to the CTF dataset could yield greater optimization performance than was observed.

A final limitation regarding the training, which happens to also be related to the dataset size, is a lack of a test data split. The train-validation split was done in this work to show that the optimization algorithm creates contexts which generalize to unseen tasks. However, there is a possibility that the algorithms were implicitly tuned to the validation data and hence the generalization might not be as great as presented in the results. If more data was available a separate test split could have been created which could have been used as a final evaluation step, after tuning the hyperparameters, such as prompts, hence avoiding the possibility of "fitting" the algorithm to the held-out data.

7.2.2 Reflections Rounds

Another limitation is the usage of only one reflection round during optimization. In the ACE paper they describe how the optimization is quite heavily improved by the usage of more reflection rounds before each update [15]. However, this was something that was not observed in our testing, which showed no improvement from using multiple reflection rounds. A potential reason for this is that, when using more than one reflection round, it was observed that the reflector ended up repeating the ground truth answer in its reflection and the generator then just used that as a source

of truth when generating its output. Hence, the reflection iteration did not steer the reflections toward better quality but only toward containing the ground truth. On the other hand, the experiments that were conducted with this were quite limited and the issue brought up could be mitigated with prompting or programmatically removing the ground truth from the reflection. Therefore it would be interesting for future work to see how the results presented in this work are affected by more reflection rounds.

7.2.3 LLM-as-judge

The usage of LLM-as-judge could be considered a limitation since they introduce noise into the optimization process. Because the reflector uses this signal to create its reflections, inaccuracies in the judges can lead to inaccurate and poor reflections. However, no experiments have been conducted that analyze the potential effect of the noise from the judges on the optimization. A direction for future work is therefore analyzing how the use of LLM judges, human evaluation and deterministic evaluations affect the performance of the optimization. On the other hand, like mentioned in Chapter 5 the variance in the judges was low in empirical testing so that impact is assumed to be small.

Another limitation related to the use of LLM-as-judge is the grounding verification. It relies on an LLM-based judge rather than deterministic verification. This could be problematic since LLMs tend to perform poorly when needing to extract a single small piece of information from a long text, which is a task that is very similar to verifying the grounding of an output from a long trajectory. Hence the judgments may be influenced by factors such as hallucinated or misleading references, as well as overly confident or well-formulated answers. This could be one of the factors that lead to the grounding performance results presented in Chapter 6 and discussed in Section 7.1.1.

7.2.4 Additional Reflector Variations

An interesting path for future work is exploring how to increase token efficiency. The more advanced configurations presented in this work, e.g. ReAct reflector, increase the token usage significantly. While they give better performance they are also more expensive and take longer to train. Therefore future work can explore how to increase the power of the reflector while maintaining the low token usage of the standard reflector.

A more concrete reflector variation that would be very interesting to explore is utilizing a retrieval augmented generation system for documentation in the reflector. This would work similar to the reflector with documentation presented in this work but with the added complexity of retrieving the correct documentation for a specific task and not storing all documentation in context at all times. This could reduce the token usage since not all documentation needs to be fed to the reflector every reflection round and could also improve generalization since it could have access to more documentation than can fit into its context window at the same time.

7.2.5 Pruning

The pruning of bad bullets in the learned skill could also be explored in greater depth in future work. The pruning strategy used in this work relied on a fixed threshold of 10 harmful tags before a section was removed. While simple and effective, this threshold was manually selected and not systematically optimized. Different thresholds may lead to different trade-offs between retaining potentially useful information and preventing harmful or outdated sections from polluting the skill. Future work could explore more adaptive pruning strategies, for example by dynamically adjusting thresholds based on task performance, section usage frequency, or the relative ratio between helpful and harmful feedback. Additionally, instead of fully removing sections, harmful sections could potentially be merged, rewritten, or temporarily disabled, similar to the maintenance strategies proposed in ACE [15].

7.2.6 Further Experiments Without Ground Truth

A limitation of the experiments without ground truth supervision is that they were only conducted for the ReAct reflector. Therefore, it is unclear whether the observed performance is a result of the ReAct reflection process or whether other reflector variants would exhibit similar behavior without ground truth. This means that it is difficult to draw broader conclusions regarding the necessity of ground truth supervision for context optimization. Future work could address this by evaluating all reflector variants without access to ground truth, which would allow for a better comparison of their ability to learn from other feedback than correctness.

7.2.7 Online Setting

Like highlighted in the results, the ReAct reflector without any ground truth supervision performs well across both datasets, especially on the search dataset. This opens up a possibility of future work, namely implementing the setup with ReAct reflector in an online setting. This can be done by collecting trajectories from the generator when actual customers to Recorded Future use it. Then feeding those trajectories to the ReAct reflector which could update the skill while the generator is online in a production environment. It would be interesting to see how this performs and if the configuration without ground truth generalizes to an online setting.

7.2.8 Same Family Bias

A potential limitation of the evaluation procedure is the use of an LLM-as-judge from the same model family as the models being evaluated. Like introduced in Chapter 3, LLM judges can have so called same-family bias, which means they assign higher scores to outputs generated by models from the same family, e.g. the GPT family by OpenAI.

However, this bias is unlikely to substantially affect the conclusions of this thesis. All evaluated agents were built upon the same underlying LLM and differed only in the skill supplied to them. As a result, any same-family bias present in the judge would

likely apply approximately equally across all evaluated configurations. While the absolute performance estimates may therefore be shifted, the relative comparisons between methods remain largely unaffected. Since the objective of this work is to compare different ACE configurations to two baselines rather than compare different model families, a consistent judge bias across all experiments does not invalidate the observed performance differences.

7.2.9 Question Categories

A final potential limitation is the use of the same seven question categories for both training and evaluation in the search dataset. Although the individual questions in the training and test sets are distinct, the optimization process may still overfit to the specific categories rather than learning more general workflows. This could lead to an overestimation of performance if the optimized skills become specialized for the types of tasks that fall within the categories. However, the categories were designed to cover the main use cases of the LLM agent and therefore should provide a reasonable approximation of the task distribution encountered in practice. On the other hand, future work could investigate the extent of this potential overfitting by splitting the categories between the training and test sets. This would allow evaluating how well optimized contexts generalize to entirely unseen categories of questions.

8

Conclusion

This thesis explored the effectiveness of ACE for improving LLM agents in domain specific and industry environments. By implementing and evaluating an optimization framework based on ACE in Recorded Future's system, the results show that context engineering is an effective method for improving the agents' performance. The optimized agents consistently outperformed a baseline with minimal context and, in most cases, also outperformed a baseline with handcrafted context made by domain experts. This demonstrates how automatic and iterative prompt refinement can rival and even outperform manual prompt engineering.

Beyond the demonstration of performance gains for the LLM agents, this work shows that the quality of reflections in ACE plays an important role in enabling learning in these niche domain environments. The proposed reflector configurations with ReAct capabilities and documentation enable higher quality feedback and hence achieve the strongest performance. These extensions demonstrate that enabling interaction with the environment or grounding reflections in domain specific knowledge can increase optimization performance. Furthermore, the competitive results without explicit ground-truth supervision suggests that learning signals can come from interaction with the environment alone, making it a relevant approach in settings where labeled data is limited.

Returning to the motivation described in the introduction, this thesis set out to explore how LLM agents can be optimized to deliver more value in real-world applications. The findings show that several important directions for future work remain, including improving the reliability of evaluation and scaling training. Yet, this work demonstrates how the LLM agents can be made more performant using methods available in real-world settings, showing how prompt optimization could be a key paradigm for the next generation of LLM agent systems.

Bibliography

[1] A. Sukharevsky et al., "The agentic organization: Contours of the next paradigm for the ai era," McKinsey & Company, London, UK, 2025, [Online]. Accessed: Jan. 26, 2026. [Online]. Available: https://www.mckinsey.com/capabilities/people-and-organizational-performance/our-insights/the-agentic-organization-contours-of-the-next-paradigm-for-the-ai-era.

[2] D. L. Poole and A. K. Mackworth, *Artificial Intelligence: Foundations of Computational Agents, 3rd edition*. Cambridge, United Kingdom: Cambridge University Press, 2023, [Online]. Accessed: Jan. 22, 2026. [Online]. Available: https://artint.info/3e/html/ArtInt3e.html.

[3] S. Mollman, "Chatgpt gained 1 million users in under a week. here's why the ai chatbot is primed to disrupt search as we know it," Yahoo! Finance, Dec. 9, 2022. Accessed: Feb. 3, 2026. [Online]. Available: https://finance.yahoo.com/news/chatgpt-gained-1-million-followers-224523258.html?guccounter=1.

[4] Anthropic, "Project vend: Can claude run a small shop? (and why does that matter?)" Anthropic PBC, USA, 2025, [Online]. Accessed: Feb. 4, 2026. [Online]. Available: https://www.anthropic.com/research/project-vend-1.

[5] Anthropic, "Project vend: Phase two," Anthropic PBC, USA, 2025, [Online]. Accessed: Feb. 4, 2026. [Online]. Available: https://www.anthropic.com/research/project-vend-2.

[6] N. Maslej et al., "Artificial intelligence index report 2025," Stanford University, USA, 2025, [Online]. Accessed: Feb. 4, 2026. [Online]. Available: https://arxiv.org/abs/2504.07139.

[7] H. Touvron et al. "Llama: Open and efficient foundation language models." [Preprint], Accessed: Feb. 3, 2026. [Online]. Available: https://arxiv.org/abs/2302.13971.

[8] H. Touvron et al. "Llama 2: Open foundation and fine-tuned chat models." [Preprint], Accessed: Feb. 3, 2026. [Online]. Available: https://arxiv.org/abs/2307.09288.

[9] X. Bi et al. "Deepseek llm scaling open-source language models with longtermism." [Preprint], Accessed: Feb. 3, 2026. [Online]. Available: https://arxiv.org/abs/2401.02954.

[10] A. Grattafiori et al. “The llama 3 herd of models.” [Preprint], Accessed: Feb. 3, 2026. [Online]. Available: https://arxiv.org/abs/2407.21783.

[11] A. Liu et al. “Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model.” [Preprint], Accessed: Feb. 3, 2026. [Online]. Available: https://arxiv.org/abs/2405.04434.

[12] A. Liu et al. “Deepseek-v3 technical report.” [Preprint], Accessed: Feb. 3, 2026. [Online]. Available: https://arxiv.org/abs/2412.19437.

[13] Meta. “The llama 4 herd: The beginning of a new era of natively multimodal ai innovation,” Accessed: Feb. 3, 2026. [Online]. Available: https://ai.meta.com/blog/llama-4-multimodal-intelligence/.

[14] D. Bergmann, “What is fine-tuning?” IBM Think, Accessed: Feb. 4, 2026. [Online]. Available: https://www.ibm.com/think/topics/fine-tuning.

[15] Q. Zhang et al., “Agentic context engineering: Evolving contexts for self-improving language models,” in *Proceedings ICLR 2026. Fourteenth International Conference on Learning Representations*, [Online], Rio de Janeiro, Brazil, 2026. Accessed: May 8, 2026. [Online]. Available: https://arxiv.org/abs/2510.04618.

[16] L. A. Agrawal et al. “Gepa: Reflective prompt evolution can outperform reinforcement learning.” [Preprint], Accessed: Jan. 26, 2026. [Online]. Available: https://arxiv.org/abs/2507.19457.

[17] R. Agarwal et al., “Many-shot in-context learning,” in *Proceedings NeurIPS 2024. Thirty-Eighth Annual Conference on Neural Information Processing Systems*, [Online], Vancouver, Canada, 2024, pp. 76 930–76 966. DOI: 10.52202/079017-2447. Accessed: Jan. 22, 2026.

[18] D. Bergmann, “What is a context window?” IBM Think, Accessed: Feb. 3, 2026. [Online]. Available: https://www.ibm.com/think/topics/context-window.

[19] S. Yao et al., “React: Synergizing reasoning and acting in language models,” in *Proceedings ICLR 2023. Eleventh International Conference on and Learning Representations*, [Online], Beijing, China, 2023. DOI: 10.48550/arXiv.2210.03629. Accessed: Jan. 22, 2026.

[20] Recorded Future. “Our story.” [Online.], Accessed: Jan. 26, 2026. [Online]. Available: https://www.recordedfuture.com/our-story.

[21] I. Belcic and C. Stryker, “What is supervised learning?” IBM Think, Accessed: Apr. 7, 2026. [Online]. Available: https://www.ibm.com/think/topics/supervised-learning.

[22] F. Lee, “What is a neural network?” IBM Think, Accessed: Apr. 7, 2026. [Online]. Available: https://www.ibm.com/think/topics/neural-networks.

[23] V. Govindaraju, V. V. Raghavan, and C. R. Rao, Eds., *Handbook of Statistics, Volume 33: Big Data Analytics*. Elsevier, 2015, [Online], ISBN: 9780444634924. Accessed: Apr. 7, 2026. [Online]. Available: https://www.sciencedirect.com/handbook/handbook-of-statistics/vol/33/suppl/C.

[24] D. Salvator, "Explaining tokens — the language and currency of ai," Nvidia blog, 2025. Accessed: Apr. 7, 2026. [Online]. Available: https://blogs.nvidia.com/blog/ai-tokens-explained/.

[25] OpenAI, *Tokenizer*, https://platform.openai.com/tokenizer, Accessed: 2026-05-12, 2026.

[26] C. Stryker, "What is a recurrent neural network (rnn)?" IBM Think, Accessed: May 6, 2026. [Online]. Available: https://www.ibm.com/think/topics/recurrent-neural-networks.

[27] A. Vaswani et al., "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon et al., Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[28] D. Bergmann and C. Stryker, "What is an attention mechanism?" IBM Think, Accessed: Apr. 8, 2026. [Online]. Available: https://www.ibm.com/think/topics/attention-mechanism.

[29] "Meta-llama/llama-3.2-1b," Huggingface, Accessed: Apr. 8, 2026. [Online]. Available: https://huggingface.co/meta-llama/Llama-3.2-1B.

[30] Mendix. "Prompt engineering," Accessed: May 20, 2026. [Online]. Available: https://docs.mendix.com/appstore/modules/genai/prompt-engineering/.

[31] Microsoft, *Advanced prompt engineering*, https://learn.microsoft.com/en-us/azure/foundry/openai/concepts/advanced-prompt-engineering?view=foundry-classic, Accessed: 2026-05-12, 2026.

[32] X. Qu et al. "A comprehensive review of ai agents: Transforming possibilities in technology and beyond." [Preprint], Accessed: Jan. 22, 2026. [Online]. Available: https://arxiv.org/abs/2508.11957.

[33] J. Luo et al. "Large language model agent: A survey on methodology, applications and challenges." [Preprint], Accessed: Apr. 22, 2026. [Online]. Available: https://arxiv.org/abs/2503.21460.

[34] Hugging Face. "What are tools?" Hugging Face Agents Course, Unit 1, Accessed: Apr. 22, 2026. [Online]. Available: https://huggingface.co/learn/agents-course/unit1/tools.

[35] Model Context Protocol. "Architecture overview." Model Context Protocol Documentation, Accessed: Apr. 22, 2026. [Online]. Available: https://modelcontextprotocol.io/docs/learn/architecture.

[36] T. Li, G. Zhang, Q. D. Do, X. Yue, and W. Chen, "Long-context LLMs struggle with long in-context learning," Transactions on Machine Learning Research, 2025, ISSN: 2835-8856. [Online]. Available: https://openreview.net/forum?id=Cw2xlg0e46.

[37] Anthropic. "Equipping agents for the real world with agent skills," Accessed: Jan. 22, 2026. [Online]. Available: https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills.

[38] Z. Shao. "Deepseekmath: Pushing the limits of mathematical reasoning in open language models." [Preprint], Accessed: Apr. 22, 2026. [Online]. Available: https://arxiv.org/abs/2402.03300.

[39] Anthropic. "Demystifying evals for ai agents," Accessed: Jan. 28, 2026. [Online]. Available: https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents.

[40] M. Chen et al. "Evaluating large language models trained on code." [Preprint], Accessed: Jan. 28, 2026. [Online]. Available: https://arxiv.org/abs/2107.03374.

[41] S. Yao, N. Shinn, P. Razavi, and K. Narasimhan. "τ-bench: A benchmark for tool-agent-user interaction in real-world domains." [Preprint], Accessed: Jan. 28, 2026. [Online]. Available: https://arxiv.org/abs/2406.12045.

[42] E. Spiliopoulou et al. "Play favorites: A statistical method to measure self-bias in llm-as-a-judge." [Preprint], Accessed: Jun. 4, 2026. [Online]. Available: https://arxiv.org/abs/2508.06709.

[43] T. B. Brown et al., "Language models are few-shot learners," CoRR, vol. abs/2005.14165, 2020. arXiv: 2005.14165. [Online]. Available: https://arxiv.org/abs/2005.14165.

[44] T. Brown et al., "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[45] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35, Curran Associates, Inc., 2022, pp. 24 824–24 837. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf.

[46] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, *Large language models are zero-shot reasoners*, 2023. arXiv: 2205.11916 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2205.11916.

[47] V. Gadesha, "What is prompt optimization?" IBM Think, Accessed: May 13, 2026. [Online]. Available: https://www.ibm.com/think/topics/prompt-optimization.

[48] ACE Team. "Ace: Agentic context engineering," GitHub, Accessed: May 20, 2026. [Online]. Available: https://github.com/ace-agent/ace.

[49] OpenAI. "Introducing gpt-5.4 mini and nano," OpenAI, Accessed: May 13, 2026. [Online]. Available: https://openai.com/index/introducing-gpt-5-4-mini-and-nano/.

A

Question Curation Prompt

```
You create evaluation data for a retrieval-augmented
search system.
Using the news article below, generate exactly TWO
high-quality question and reference answer pairs.

Target question styles:
- Question 1 style id: {question_style_1}
- Question 1 style guidance: {style_guidance_1}
- Question 2 style id: {question_style_2}
- Question 2 style guidance: {style_guidance_2}

Goal alignment:
- Each question must be grounded in this article's event.
- Each question should NOT require this exact article to answer.
- It should be answerable from other credible coverage of the same event.
- Avoid article-dependent phrasing like "according to this article/report."

Question quality:
- Make it realistic and resemble what a real user asks
  a search system.
- Follow the target question style for each question.
- Make it moderately hard, but let the difficulty come
  from the selected style.
- Some questions can be two-slot factual questions,
  while others can focus on cause, scope, distinction,
  timeline, entity resolution, or stable numeric
  details.
- Avoid generic questions like "What happened?".
- Avoid yes/no questions.
- Numeric questions are allowed sometimes, but not
  every time.
- Use numeric details only when they are central to the
  event and likely stable across multiple sources.
- The two questions must be meaningfully different.
- They must not be paraphrases, minor rewrites, or
  simple slot-swaps of the same question.
- Prefer two different angles on the same event,
  especially when the sampled styles differ.
Reference answer quality:
- The answer must be factual, concise, and complete
  enough to judge correctness semantically.
- Use neutral paraphrasing, not article-specific
  wording.
- Include the key relationships/facts needed to answer
  the question well.
- If numbers are uncertain across sources, prefer
  cautious wording such as "about", "at least", or
  "around" instead of brittle exactness.
- Do not invent facts.
Return valid JSON only.
Format:
{
  "questions": [
    {
      "question": "...",
      "answer": "...",
      "topic": "geopolitics | war | cybersecurity",
      "question_style": "{question_style_1}"
    },
    {
      "question": "...",
      "answer": "...",
      "topic": "geopolitics | war | cybersecurity",
      "question_style": "{question_style_2}"
    }
  ]
}
ARTICLE:
{article_text}
```

B

Judge Prompts

B.1 Correctness Judge for Search Dataset

```
Evaluate whether the model answer should PASS or FAIL
as a lenient expected-answer alignment check.
This is NOT a strict factual correctness scorer.
Use the expected response only as a reference for what
topic, event, actor, entity, relationship, or fact
pattern the answer was supposed to address.

Judge only the final answer content.
Use semantic/event matching, not exact wording.

Core principle:
PASS if the answer is clearly addressing the same
underlying question, event, story, actor, entity, or
requested comparison as the expected response, even when
some factual details are wrong, swapped, incomplete,
oversimplified, or misinterpreted.

FAIL if the answer drifts to a materially different
event, campaign, organization, actor, incident, or
storyline, even if it sounds superficially similar.

Decision rubric:
PASS = same subject/event/question is clearly being
addressed; factual misalignment is allowed.

FAIL = different subject/event/question, empty answer,
or only a weak/speculative topical overlap.

Important rules:
- Do NOT require the answer to be factually correct in
  order to pass.
- Do NOT fail just because the final named person,
  company, date, number, or side of a comparison is
  wrong, as long as the answer is still clearly about
  the same event or question.
- If a multi-part answer gets some parts wrong but is
  unmistakably discussing the same incident or
  comparison, prefer PASS.
- If the answer uses the wrong source interpretation but
  still refers to the same event asked about, prefer
  PASS.
- If the answer mainly describes a different campaign,
  different victim set, different country, or different
  intrusion chain than the one asked about, FAIL.
- If the answer is so generic that you cannot tell
  whether it refers to the same event/question, FAIL.
- Prefer leniency when uncertain between pass and fail.

In-context examples:

Example 1:
Question:
How did UConn's status entering the NCAA title game
differ between its 2024 championship run and the later
championship matchup against Michigan?

Expected response:
In 2024, UConn entered as a favorite pursuing a repeat.
Later against Michigan, UConn was an underdog chasing a
three-titles-in-four-years milestone.

Model answer:
Discusses the same comparison and title matchup, but
misstates details and milestone.

Result: PASS.
```

```
Why:
Same storyline and comparison despite factual mistakes.

Example 2:
Question:
Who declared that South Korea would remain North
Korea's most hostile enemy state?

Expected response:
Jang Kum Chol.

Model answer:
Kim Jong Un.

Result: PASS.

Why:
Wrong actor, but clearly the same event/question.
Example 3:
Question:
In a GitHub-based intrusion campaign attributed to
North Korean actors, what organizations were targeted
and what systems/services were used or at risk?

Expected response:
South Korean organizations; Windows endpoints; GitHub
repos/accounts used as C2 and exfiltration channels.

Model answer:
Focuses mainly on JumpCloud and a separate GitHub
campaign.

Result: FAIL.
Why:
Different intrusion storyline.
Example 4:
Question:
Which threat actor compromised LiteLLM packages on PyPI
in March 2026?
Expected response:
TeamPCP.
Model answer:
UNC1069.

Result: FAIL.
Why:
Single-entity identification questions usually fail
when a different actor is named.
Example 5:
Question:
In the Iran-linked campaign against U.S. critical
infrastructure, what facilities were hit and what
industrial component was targeted?

Expected response:
Oil/gas sites and water facilities; internet-facing
PLCs.

Model answer:
Discusses water facilities and PLC-controlled pumps.

Result: PASS.
Why:
Clearly the same campaign and attack mechanism despite
missing details.
Special rule for single-entity identification questions:
- If the question is essentially "who/which actor was
  it?" and the answer names a different entity than the
  expected response, that usually FAILS.
- Exception: PASS if the answer is clearly an alias or
  alternate rendering of the same entity.
- If the question is broader than pure identification
  and the answer clearly discusses the same event while
  misidentifying one participant, prefer PASS.

Question:
{question}

Expected response:
{expected_response}

Model answer:
{answer_text}

Return valid JSON only in this shape:
{"result": "pass", "rationale": "short explanation"}

Use only "pass" or "fail" for result.
```

B.2 Correctness Judge for CTF Dataset

```
Evaluate whether the model answer should PASS or FAIL
for a keyword-oriented expected-answer check.
This dataset's expected responses are often imperfect
keywords, aliases, fragments, or approximate target
strings. Use them as strong clues, not as a brittle
exact-match requirement.
Judge only the final answer content.

Core principle:
PASS if the answer likely contains, identifies, or
expresses the intended expected answer, even when the
dataset keyword is slightly imperfect, abbreviated,
aliased, formatted differently, or only approximately
phrased.

FAIL if the answer clearly does not contain the intended
answer, points to a different answer, or is too vague
to reasonably count.

How to use keyword groups:
- Comma-separated groups usually indicate multiple
  expected answer parts.
- Slash-separated items within a group are candidate
  variants or aliases.
- Exact presence is helpful evidence, but absence of an
  exact string is NOT automatic failure.
- If the answer contains a near match, obvious alias,
  formatting variant, or semantically equivalent answer,
  you may still PASS.
- If multiple groups represent distinct required parts,
  prefer PASS when the answer substantially covers the
  intended answer parts, even if one group is slightly
  noisy or imperfectly represented.

Leniency rules:
- Prefer PASS when the answer is very likely pointing to
  the intended entity/value despite imperfect dataset
  keywords.
- Do not require the keyword to be the main focus of the
  response.
- If a correct value appears anywhere in the output,
  that is strong evidence for PASS.
- Treat punctuation, casing, spacing, hyphenation, and
  minor wording differences as irrelevant.
- If the output lists several options but includes the
  likely intended expected answer, you may still PASS.
- FAIL only when the intended answer is missing,
  contradicted, or replaced by a clearly different
  answer.
- Prefer PASS when uncertain.

Question:
{question}

Expected keyword groups:
{keyword_groups}

Matched keyword variants found in answer:
{matched_keywords}

Model answer:
{answer_text}

Return valid JSON only in this shape:
{"result": "pass", "rationale": "short explanation"}

Use only "pass" or "fail" for result.
```

B.3 Grounding Judge

```
You are an expert evaluator. Assess whether the model''s answer is consistent with verified
     information gathered from tool calls. Evaluation criteria: 1. Contextual relevance - Does the
     answer reflect the circumstances described in the fetched information without introducing
     unrelated information? 2. Hallucination check - Does the response avoid adding facts not
     mentioned in the data from the tools? - Is the response grounded in what data retreived by the
     model?

Evaluate groundedness and evidence alignment only.
Do not score against the expected response here.
Return PASS or FAIL.

Be reasonably strict. Do not reward fluent wording,
plausible synthesis, or generic risk language if the
answer's main claims are unsupported by retrieved evidence.

Decision rubric:
PASS = the main claims in the answer are supported by the
retrieved trajectory/context, with at most minor omission,
uncertainty, or trivial overreach.

FAIL = the answer is mostly unsupported, fabricated,
contradictory to the retrieved evidence, or overclaims
beyond what the evidence shows.

Important rules:
- If the answer states material facts not supported by
  trajectory/context, prefer FAIL.
- If evidence is insufficient and the model clearly states
  uncertainty, prefer PASS over unsupported guessing.
- If the answer goes meaningfully beyond what the retrieved
  evidence shows, FAIL.
- If the answer makes a confident claim where the retrieved
  evidence is only partial or ambiguous, FAIL.
- Prefer FAIL when uncertain.

Question:
{question}

Retrieved trajectory/context:
{trajectory}

Output:
{response_text}

Return valid JSON only in this shape:
{"result": "pass", "rationale": "short explanation"}

Use only "pass" or "fail" for result.
```

C

Standard Reflector

C.1 Search and CTF Dataset

The following prompt was used for the reflector when optimizing for the search dataset and the CTF dataset.

```
You are an expert analyst and educator. Diagnose why the model's attempt was wrong or incomplete by comparing trajectory, predicted
      answer, expected answer, and environment feedback.

Instructions:
- Identify what specifically failed, why it failed, and what should be done differently next time.
- Focus on root causes, not surface wording.
- Prioritize actionable lessons that improve:
  1) tool usage and selection (which tool to call, what to retrieve with it, and when to call it)
  2) search workflow (resolution, source selection, verification, stopping criteria)
  3) evidence-grounded synthesis (especially multi-part questions)
- Do not overfit to the current question; keep guidance reusable.
- If the failure is query quality, propose one corrected placeholder TQL example in 'correct_approach' (for example:
      [redacted] and
      explain why it is better.
- Keep each text field concise (1-3 short sentences).
bullet_tags rules:
- Tag only sections that were materially causal.
- Use ONLY section IDs that exist in the provided section list.
- Never invent IDs.
- Section id format: 'Section Name' (main skill only).
- Keep non-neutral tags limited (max 4).

Answer in this exact JSON format:
{{
  "reasoning": "brief analysis",
  "error_identification": "what went wrong",
  "root_cause_analysis": "why it went wrong",
  "correct_approach": "what should be done instead",
  "key_insight": "reusable principle to remember",
  "bullet_tags": [
    {{"id": "Section Name", "tag": "helpful|harmful|neutral"}}
  ]
}}

## INPUTS

### Question:
{}

### Model's Predicted Answer:
{}

### Expected/Correct Answer:
{}

### Environment/Judge Feedback:
{}

### Part of Skill Used by the Generator:
{}

### Model's Trajectory:
{}
```

D

Reflector With docs

D.1 Search and CTF Dataset

The following prompt was used for the reflector when optimizing for the search dataset and the CTF dataset.

```
You are a careful tool-focused reviewer. Your job is to diagnose why the model failed and convert that failure into precise, reusable
      , doc-backed lessons for improving Recorded Future search.

Primary objective: identify the exact search failure in tool/operator/filter/workflow terms and produce the smallest reusable
      correction that would improve future tool usage.

You have two sources of evidence:
1. The model trajectory (tool calls and results)
2. Relevant TQL documentation

Use both directly. Do not guess.

---

### CORE RULES

- Be concrete. Name exact tools, operators, filters, or query shapes.
- Focus on the FIRST mistake that caused the failure.
- Do NOT write long explanations.
- Do NOT restate generic advice (e.g. "open the full document") unless a specific tool/operator/filter mistake is involved.
- Prefer one strong correction over many weak ones.

---

### FAILURE TAXONOMY

You may use one or more of these labels internally while reasoning, but do NOT return them as separate JSON fields:

- bad_tool_choice
- bad_query_family
- bad_operator_choice
- bad_filter_structure
- bad_tql_syntax
- bad_entity_usage
- bad_source_or_media_selection
- bad_document_resolution
- bad_stopping_rule
- bad_evidence_boundary
- bad_cross_document_synthesis
- bad_comparison_workflow

---

### PRIMARY LESSON FAMILY

You may choose one of these internally to sharpen the lesson, but do NOT return it as a separate JSON field:

- exact_article_resolution
- title_slug_anchor_usage
- keyword_search_pivot
- reference_tql_shape
- entity_id_pivot
- published_filter_usage
- comparison_separate_docs
- stop_rule_and_pivot
- syntax_validation
- evidence_boundary_enforcement
- avoid_adjacent_synthesis

---
```

```
### WHAT TO CHECK IN THE TRAJECTORY
You MUST identify at least TWO concrete issues:

- wrong or missing tool (e.g. did not use [redacted]
- bad query shape (long natural-language '[redacted]', bad 'and/or')
- wrong operator ([redacted] vs [redacted], missing [redacted])
- missing or incorrect filters ('published', etc.)
- failure to pivot after 0 / huge / noisy results
- failure to resolve the correct document_id
- answering from wrong or adjacent document
- mixing multiple documents into one answer
- whether the source/media/entity type matched the questions content class

---

### USING THE TQL DOCS

- If a better tool/operator/filter is implied by the docs, state it.
- Include small examples when helpful.
- If exact values would help (e.g. [redacted] include them.

SOURCE / MEDIA / ENTITY SELECTION:
- If the question implies a specific content class (platform, ecosystem, event type), check whether the correct TQL dimensions were
      used:
[redacted]
- If the model searched too broadly instead of constraining these, call it out.
- Only apply this when clearly relevant.

---

### STOP RULE

You MUST state when the model should have pivoted or stopped.

Examples:
- 0 or huge counts  wrong query shape
- noisy results  switch query family
- wrong document opened  rerun with better anchor

---

### NEVER DO THIS AGAIN

State one clear rule the model must not repeat.

Example:
- Never answer from a broad discovery hit instead of a resolved document.

---

### OUTPUT FORMAT (STRICT JSON)

Return ONLY valid JSON:

JSON safety rules:
- Return exactly one JSON object and no surrounding commentary.
- Use exactly these keys and no others: 'reasoning', 'error_identification', 'root_cause_analysis', 'correct_approach', 'key_insight'
      , 'bullet_tags'.
- Keep all five text fields short and simple.
- If you include a tool or TQL example inside a string field, keep it JSON-safe: prefer plain English or single quotes, and avoid
      unescaped double quotes inside the string value.
- Do NOT add extra fields such as 'failure_modes' or 'primary_lesson_family'.

Answer in this exact JSON format:
{{
  "reasoning": "brief analysis",
  "error_identification": "what went wrong",
  "root_cause_analysis": "why it went wrong",
  "correct_approach": "what should be done instead",
  "key_insight": "reusable principle to remember",
  "bullet_tags": [
    {{"id": "Section Name", "tag": "helpful|harmful|neutral"}}
  ]
}}

---

### BULLET TAGGING RULES

- Tag only provided sections.
- helpful = directly teaches the fix
- neutral = unrelated or too generic
- harmful = actively pushes wrong behavior

IMPORTANT:
- Do NOT mark something harmful just because it didn't prevent the error.
```

```
---

### QUALITY CHECK (FINAL)

Before answering, ensure:

- at least 2 concrete failures identified
- exact tool/operator/filter mentioned
- one clear correction
- one clear stop rule
- no generic filler

---

### INPUTS

Question:
{}

Model's Predicted Answer:
{}

Expected/Correct Answer:
{}

Environment/Judge Feedback:
{}

Part of Skill Used by the Generator:
{}

Relevant TQL Documentation:
{}

Model's Trajectory:
{}
"""
```

E

ReAct Reflector Prompts

E.1 Search and CTF Dataset with Ground Truth

The following prompt was used for the reflector when optimizing for the search dataset and the CTF dataset.

```
You are an expert on Recorded Future search, TQL, query debugging, and exploratory search strategy.

You are a ReAct-style reflector. Your role is to inspect the generator's trajectory, identify weak TQL usage, and then aggressively
      explore the TQL space with tools to discover better patterns, constraints, and examples.

Important distinction:
- The provided trajectory is the GENERATOR'S historical trajectory, not your own work in this reflector session.
- The generator trajectory is evidence to inspect, reproduce, challenge, and improve.
- Do NOT treat generator tool calls, generator experiments, or generator results as if you personally executed them in this reflector
       run.
- Your own experiments are only the tool calls you make now as the reflector.

Tooling assumption:
- You have access to the same Recorded Future tools and core search capabilities as the generator.
- For this task setup, assume the needed underlying data is available through the tools if TQL and tool usage are done correctly.
- If relevant references, entities, or documents are not found, treat that primarily as evidence that the query, filter composition,
      entity resolution, tool choice, or search workflow can be improved.
- Do not accept "the data was unavailable" as the default explanation when a stronger TQL/query reformulation or better tool sequence
       could plausibly recover it.
- If a better outcome depended on choosing a different tool, calling tools in a different order, or using the same tool more
      effectively, say that explicitly in the reflection.
- When helpful, name the concrete tool or tool family the generator should have used or should have used earlier.
- Use tool choice as part of the lesson when the failure was not only about TQL syntax but also about search workflow, entity
      resolution, count-first validation, or document retrieval strategy.

Your end goal is to improve the TQL documentation used by the generator.
You are not only critiquing an answer. You are acting like an experimental TQL researcher whose job is to probe the search language
      and uncover better ways to search.

Exploratory TQL querying is mandatory, not optional.
Before you finalize, you must use tools to run exploratory TQL investigation whenever the task involves search quality, query
      formulation, retrieval failure, weak recall/precision, entity resolution, query debugging, or uncertainty about a better search
       path.
In normal cases, you should test multiple nearby TQL variants instead of relying on a single query attempt.
Minimum experimentation requirement:
- Before finalizing, run at least 3 tool-backed TQL experiments yourself unless the provided trajectory already contains equivalent
      experiments with clear outcomes and you explicitly label them as inherited generator evidence.
- At least 1 experiment must test a risky, uncertain, malformed-looking, or potentially invalid TQL assumption if syntax/field/
      operator uncertainty is part of the diagnosis.
- At least 1 experiment must be broader than the generator's attempted query.
- At least 1 experiment must be more structured, constrained, or entity-aware than the generator's attempted query.
- If you have not run these experiments, you are not ready to finalize.

You have only two valid actions at any step:
1. Call a tool to investigate, reproduce, test, or stress a TQL idea.
2. Call [redacted] with the final reflection JSON in the exact required format.

If you are not yet ready to call [redacted] with the final JSON reflection, call a tool.
Do not produce any other kind of free-form assistant response.

Primary objective:
Produce a concise reflection that improves future TQL documentation for the generator by identifying:
- what queries or search steps in the trajectory were bad, weak, malformed, too timid, or incomplete
- what improved query shapes or workflows work better
- what TQL constraints, syntax discoveries, lookup patterns, or debugging lessons should be documented
- What TQL search workflows are highly optimized for search questions


Exploration mindset:
- Be highly exploratory.
```

```
- TQL is very similar to [redacted]
- Treat this as a mini TQL lab. Probe syntax, query shape, field usage, entity resolution patterns, timeframe filters, and count-
      first workflows.
- Assume weak retrieval usually means the search can be improved. Zero results, noisy results, or missing evidence are usually
      debugging signals, not stopping conditions.
- The default behavior is to explore with TQL tools before concluding. Reflection without exploratory querying is usually incomplete.
- When the generator used TQL poorly or too little, compensate by running the exploratory queries the generator should have tried.
- Wild experiments are allowed if they are still relevant to diagnosing the failure.
- Do not be afraid of tool errors. Unsupported syntax, invalid fields, malformed query shapes, and empty-result paths are all useful
      evidence.
- Failed tool calls are informative because they reveal what TQL can and cannot do.
- If the generator stopped after one weak query path, you should usually explore several nearby alternatives before finalizing.

Decision rule:
- If any part of your diagnosis depends on whether information was retrievable, whether the query was too broad or too narrow,
      whether the syntax/shape was wrong, whether the wrong entity or entity type was used, whether a count-first workflow was
      missing, or whether another TQL shape works better, call a tool.
- If the task is about Recorded Future search or TQL, assume exploratory tool use is required unless the trajectory already contains
      enough verified experiments to support the conclusion.
- Do not treat the generator trajectory alone as sufficient just because it contains tool calls; if the generator's exploration was
      weak, incomplete, or inconclusive, you should run your own reflector experiments.
- If the generator failed to retrieve relevant evidence, assume there is likely a better TQL or tool path to find it and keep probing
       until you have a strong explanation for what was wrong.
- If unsure whether to finalize or continue exploring, continue exploring.
- Prefer more experimentation when the trajectory suggests uncertainty, generic searching, malformed TQL, weak structure, or
      premature stopping.
- Only output final JSON when the failure mode and the better alternatives are well-supported by exploratory tool evidence.
- Do not finalize immediately after one repair attempt if there are still plausible nearby query variants worth testing.
- If there is any uncertainty about whether a syntax, field, operator, nesting pattern, or query family is valid, test it directly
      with a tool instead of reasoning it away.
- If a suspicious or invalid-looking TQL pattern might plausibly work, you should usually try it once and use the tool result as
      evidence.

What to investigate with tools:
- Recreate or approximate the generator's search path when the trajectory shows a concrete attempt.
- Treat the generator trajectory as a baseline to reproduce and improve, not as proof that you already explored enough.
- Identify bad queries and explain why they were bad:
  too abstract, too broad, too narrow, wrong entity, wrong timeframe, wrong filter composition, weak query structure, invalid syntax,
       unsupported field assumptions, or missing count/refinement.
- Try several improved variants when useful, not just one.
- For most TQL failures, test at least one broader variant and one more structured or constrained variant before finalizing.
- Test broader, narrower, more structured, and more entity-driven alternatives when that comparison helps explain the failure.
- When syntax is uncertain, deliberately probe risky alternatives such as different nesting, field usage, operator placement, or
      simplified/minimal query shapes.
- It is acceptable and often useful to intentionally try malformed-looking or speculative TQL if it helps map what the system accepts
       or rejects.
- Use [redacted] frequently to cheaply test viability.
- Use retrieval tools when you need to compare result quality, not just count.
- Use [redacted] when entity resolution, entity type, or entity ID usage may be the missing step.
- Try deliberately different query families if the first repair still looks weak.
- If a query throws an error, capture what that error teaches about valid vs invalid TQL assumptions.
- Use multiple rounds to explore the tools and TQL syntax, these explorations and reflections will help downstream reflection and
      curator to get a high value tql documentation.

Tool policy:
- Use tools for verification, count checks, entity resolution, and aggressive TQL experimentation.
- Do NOT use tools to browse for extra skill content. The relevant skill context is already provided.
- Prefer tool-backed judgments over intuition.
- Prefer exploratory TQL query evidence over purely verbal critique.
- Do not avoid a tool call because it may fail.
- Do not avoid a speculative or invalid-looking TQL test just because it may error; informative failures are valuable evidence.
- Do not invent tool results.
- Do not invent TQL syntax, fields, or operators. If a query shape is uncertain, test it. Retreive feedback in the form of counds,
      syntax errors or weak results
- Stop once you have enough evidence to explain both the failure and the better reusable pattern.

Reflection instructions:
- Focus first on the generator's search trace and query behavior, not just the final answer wording.
- Explicitly identify which search actions or query shapes were bad.
- Explain how those queries could be improved.
- Say directly if the generator should have explored more instead of stopping early.
- Be explicit about what TQL worked and what did not work.
- Name specific failed query shapes, invalid syntax attempts, weak query families, or zero-result patterns when they mattered.
- Name specific successful or better-performing query shapes when they mattered.
- If you verified an improved query that materially outperformed the generator's attempted path, include the concrete improved TQL
      query or query dict in 'correct_approach' and clearly say why it was better.
- When useful, contrast the failing query pattern against the better query pattern instead of describing only one side.
- When tool choice mattered, explicitly name which tool the generator should have used, should have used first, or should have paired
       with the better TQL query.
- Highlight tool usage patterns that should become part of the generator's reusable workflow, especially around
      [redacted]
- If multiple experiments mattered, summarize the best one or two findings, not every trial.
- If an experiment failed but taught an important constraint, mention the failed pattern and what it revealed.
- Explicitly mention when you tested a speculative or invalid-looking query shape and what the tool response taught you.
- Prefer concrete examples over abstract descriptions when reporting TQL findings.
- Favor lessons that can become TQL documentation for the generator:
  1) query shape rules
  2) syntax/field constraints
  3) entity resolution and entity ID usage
  4) count-first and refine workflows
  5) debugging empty, noisy, or invalid searches
  6) exploration habits for discovering better query shapes
- Keep the lesson reusable rather than question-specific.
```

```
Documentation-oriented lens:
- Your reflection should help a curator write stronger TQL documentation for the generator.
- Prefer surfacing concrete findings that can be documented:
  example TQL snippets, syntax constraints, query-shape patterns, and compact lookup tables.
- If the real lesson is about a TQL lookup table the generator needs, say so explicitly.
- If the real lesson is about a family of examples the skill is missing, say so explicitly.

bullet_tags rules:
- Tag only sections that were materially causal.
- Use ONLY section IDs that exist in the provided section list.
- Never invent IDs.
- Section id format: ‘Section Name‘.
- Keep non-neutral tags limited (max 4).
- If the failure is mainly a misapplication of an existing section, tag that section rather than implying a totally new lesson by
      default.

Question:
{}

## Model’s Reasoning Trace/Trajectory:
{}

This section above is generator history only. It is not part of your reflector session history.

---

Model’s Predicted Answer:
{}

Expected/Correct Answer:
{}

Environment Feedback:
{}

Part of Skill Used by the Generator:
{}

Domain knowledge:
- The generator has no prior knowledge of TQL syntax or best practices.
- The generator must be taught through documentation, examples, lookup tables, and reusable query patterns.
- Infer lessons only from the provided trajectory, skill context, verified tool evidence, and reference manual.
- Do not invent TQL syntax, operators, or undocumented patterns.
- Use the current skill sections to judge novelty: suggest only the missing reusable lesson or the repair to a harmful section.

TQL reference manual:
{}

Final output rules:
- Do not output the final JSON as a normal assistant message.
- When you are done, call [redacted] and pass the final JSON object as its payload.
- The payload passed to [redacted] must be JSON only.
- Do not include markdown fences.
- Do not include commentary before or after the JSON.
- The JSON must be grounded in exploratory TQL tool use, not just trajectory reading.
- Until you are ready to call [redacted] with that final JSON, your next action should be a tool call.

Answer in this exact JSON format:
{{
  "reasoning": "brief analysis grounded in the trace and verified experiments",
  "error_identification": "which search/query or tool choices were bad, what specific TQL did not work, and what failed",
  "root_cause_analysis": "why those queries or choices failed and what TQL misunderstanding caused it",
  "correct_approach": "what better TQL/query and tool workflow should be tried and documented, explicitly highlighting the better-
       performing query or query family, the relevant tools to use, and including a concrete improved TQL query when one was
       verified to outperform the generator path",
  "key_insight": "the reusable TQL or search principle the generator documentation should teach",
  "bullet_tags": [
    {{"id": "Section Name", "tag": "helpful|harmful|neutral"}}
  ]
}}
---
```

E.2 Search Dataset without Ground Truth

The following prompt was used for the reflector when optimizing for the search dataset without ground truth supervision.

```
You are an expert on Recorded Future search, TQL, query debugging, and exploratory search strategy.

You are a ReAct-style reflector. Your role is to inspect the generator’s trajectory, identify weak TQL usage, and then aggressively
      explore the TQL space with tools to discover better patterns, constraints, and examples.

Important distinction:
- The provided trajectory is the GENERATOR’S historical trajectory, not your own work in this reflector session.
- The generator trajectory is evidence to inspect, reproduce, challenge, and improve.
- Do NOT treat generator tool calls, generator experiments, or generator results as if you personally executed them in this reflector
       run.
```

```
- Your own experiments are only the tool calls you make now as the reflector.

Tooling assumption:
- You have access to the same Recorded Future tools and core search capabilities as the generator.
- For this task setup, assume the needed underlying data is available through the tools if TQL and tool usage are done correctly.
- If relevant references, entities, or documents are not found, treat that primarily as evidence that the query, filter composition,
      entity resolution, tool choice, or search workflow can be improved.
- Do not accept "the data was unavailable" as the default explanation when a stronger TQL/query reformulation or better tool sequence
       could plausibly recover it.
- If a better outcome depended on choosing a different tool, calling tools in a different order, or using the same tool more
      effectively, say that explicitly in the reflection.
- When helpful, name the concrete tool or tool family the generator should have used or should have used earlier.
- Use tool choice as part of the lesson when the failure was not only about TQL syntax but also about search workflow, entity
      resolution, count-first validation, or document retrieval strategy.

Correctness judgment:
- Environment feedback may no longer contain an explicit correctness verdict.
- You must infer whether the generator's predicted answer is correct from the question, the predicted answer, the expected/correct
      answer reference, the generator trajectory, and your own tool-backed investigation.
- Treat environment feedback as auxiliary context only. It may contain grounding or process feedback, but it is not the source of
      truth for correctness.
- Do not assume the answer is correct or incorrect just because the environment feedback is neutral, sparse, or missing correctness
      language.
- If correctness is uncertain, investigate with tools until you can explain whether the search path likely supported the answer or
      missed better evidence.


Your end goal is to improve the TQL documentation used by the generator.
You are not only critiquing an answer. You are acting like an experimental TQL researcher whose job is to probe the search language
      and uncover better ways to search.

Exploratory TQL querying is mandatory, not optional.
Before you finalize, you must use tools to run exploratory TQL investigation whenever the task involves search quality, query
      formulation, retrieval failure, weak recall/precision, entity resolution, query debugging, or uncertainty about a better search
       path.
In normal cases, you should test multiple nearby TQL variants instead of relying on a single query attempt.
Minimum experimentation requirement:
- Before finalizing, run at least 3 tool-backed TQL experiments yourself unless the provided trajectory already contains equivalent
      experiments with clear outcomes and you explicitly label them as inherited generator evidence.
- At least 1 experiment must test a risky, uncertain, malformed-looking, or potentially invalid TQL assumption if syntax/field/
      operator uncertainty is part of the diagnosis.
- At least 1 experiment must be broader than the generator's attempted query.
- At least 1 experiment must be more structured, constrained, or entity-aware than the generator's attempted query.
- If you have not run these experiments, you are not ready to finalize.

You have only two valid actions at any step:
1. Call a tool to investigate, reproduce, test, or stress a TQL idea.
2. Call [redacted] with the final reflection JSON in the exact required format.

If you are not yet ready to call [redacted] with the final JSON reflection, call a tool.
Do not produce any other kind of free-form assistant response.

Primary objective:
Produce a concise reflection that improves future TQL documentation for the generator by identifying:
- what queries or search steps in the trajectory were bad, weak, malformed, too timid, or incomplete
- what improved query shapes or workflows work better
- what TQL constraints, syntax discoveries, lookup patterns, or debugging lessons should be documented
- what TQL search workflows are highly optimized for search questions
- whether the generator's final answer was actually supported by the search path and evidence it found


Exploration mindset:
- Be highly exploratory.
- TQL is very similar to [redacted]
- Treat this as a mini TQL lab. Probe syntax, query shape, field usage, entity resolution patterns, timeframe filters, and count-
      first workflows.
- Assume weak retrieval usually means the search can be improved. Zero results, noisy results, or missing evidence are usually
      debugging signals, not stopping conditions.
- The default behavior is to explore with TQL tools before concluding. Reflection without exploratory querying is usually incomplete.
- When the generator used TQL poorly or too little, compensate by running the exploratory queries the generator should have tried.
- Wild experiments are allowed if they are still relevant to diagnosing the failure.
- Do not be afraid of tool errors. Unsupported syntax, invalid fields, malformed query shapes, and empty-result paths are all useful
      evidence.
- Failed tool calls are informative because they reveal what TQL can and cannot do.
- If the generator stopped after one weak query path, you should usually explore several nearby alternatives before finalizing.

Decision rule:
- If any part of your diagnosis depends on whether information was retrievable, whether the query was too broad or too narrow,
      whether the syntax/shape was wrong, whether the wrong entity or entity type was used, whether a count-first workflow was
      missing, or whether another TQL shape works better, call a tool.
- If the task is about Recorded Future search or TQL, assume exploratory tool use is required unless the trajectory already contains
      enough verified experiments to support the conclusion.
- Do not treat the generator trajectory alone as sufficient just because it contains tool calls; if the generator's exploration was
      weak, incomplete, or inconclusive, you should run your own reflector experiments.
- If the generator failed to retrieve relevant evidence, assume there is likely a better TQL or tool path to find it and keep probing
       until you have a strong explanation for what was wrong.
- If unsure whether to finalize or continue exploring, continue exploring.
- Prefer more experimentation when the trajectory suggests uncertainty, generic searching, malformed TQL, weak structure, or
      premature stopping.
- Only output final JSON when the failure mode and the better alternatives are well-supported by exploratory tool evidence.
- Do not finalize immediately after one repair attempt if there are still plausible nearby query variants worth testing.
- If there is any uncertainty about whether a syntax, field, operator, nesting pattern, or query family is valid, test it directly
      with a tool instead of reasoning it away.
- If a suspicious or invalid-looking TQL pattern might plausibly work, you should usually try it once and use the tool result as
```

```
      evidence.

What to investigate with tools:
- Recreate or approximate the generator's search path when the trajectory shows a concrete attempt.
- Treat the generator trajectory as a baseline to reproduce and improve, not as proof that you already explored enough.
- Identify bad queries and explain why they were bad:
  too abstract, too broad, too narrow, wrong entity, wrong timeframe, wrong filter composition, weak query structure, invalid syntax,
       unsupported field assumptions, or missing count/refinement.
- Try several improved variants when useful, not just one.
- For most TQL failures, test at least one broader variant and one more structured or constrained variant before finalizing.
- Test broader, narrower, more structured, and more entity-driven alternatives when that comparison helps explain the failure.
- When syntax is uncertain, deliberately probe risky alternatives such as different nesting, field usage, operator placement, or
      simplified/minimal query shapes.
- It is acceptable and often useful to intentionally try malformed-looking or speculative TQL if it helps map what the system accepts
       or rejects.
- Use [redacted] frequently to cheaply test viability.
- Use retrieval tools when you need to compare result quality, not just count.
- Use [redacted] when entity resolution, entity type, or entity ID usage may be the missing step.
- Try deliberately different query families if the first repair still looks weak.
- If a query throws an error, capture what that error teaches about valid vs invalid TQL assumptions.
- Use multiple rounds to explore the tools and TQL syntax; these explorations should support both the reflection and the downstream
      TQL documentation.
- If the generator's final answer may be unsupported, explicitly test whether the answer's core claim is actually retrievable through
       a better search path.

Tool policy:
- Use tools for verification, count checks, entity resolution, and aggressive TQL experimentation.
- Do NOT use tools to browse for extra skill content. The relevant skill context is already provided.
- Prefer tool-backed judgments over intuition.
- Prefer exploratory TQL query evidence over purely verbal critique.
- Do not avoid a tool call because it may fail.
- Do not avoid a speculative or invalid-looking TQL test just because it may error; informative failures are valuable evidence.
- Do not invent tool results.
- Do not invent TQL syntax, fields, or operators. If a query shape is uncertain, test it. Retreive feedback in the form of counds,
      syntax errors or weak results
- Stop once you have enough evidence to explain both the failure and the better reusable pattern.

Reflection instructions:
- Focus first on the generator's search trace and query behavior, not just the final answer wording.
- Explicitly identify which search actions or query shapes were bad.
- Explain how those queries could be improved.
- Say directly if the generator should have explored more instead of stopping early.
- Be explicit about what TQL worked and what did not work.
- Name specific failed query shapes, invalid syntax attempts, weak query families, or zero-result patterns when they mattered.
- Name specific successful or better-performing query shapes when they mattered.
- If the predicted answer was unsupported, incomplete, or contradicted by the expected answer or tool-backed evidence, say that
      explicitly.
- If you verified an improved query that materially outperformed the generator's attempted path, include the concrete improved TQL
      query or query dict in 'correct_approach' and clearly say why it was better.
- When useful, contrast the failing query pattern against the better query pattern instead of describing only one side.
- When tool choice mattered, explicitly name which tool the generator should have used, should have used first, or should have paired
       with the better TQL query.
- Highlight tool usage patterns that should become part of the generator's reusable workflow, especially around
  [redacted]
- If multiple experiments mattered, summarize the best one or two findings, not every trial.
- If an experiment failed but taught an important constraint, mention the failed pattern and what it revealed.
- Explicitly mention when you tested a speculative or invalid-looking query shape and what the tool response taught you.
- Prefer concrete examples over abstract descriptions when reporting TQL findings.
- Favor lessons that can become TQL documentation for the generator:
  1) query shape rules
  2) syntax/field constraints
  3) entity resolution and entity ID usage
  4) count-first and refine workflows
  5) debugging empty, noisy, or invalid searches
  6) exploration habits for discovering better query shapes
  7) validating whether a final answer is actually supported by retrieved evidence
- Keep the lesson reusable rather than question-specific.

Documentation-oriented lens:
- Your reflection should help a curator write stronger TQL documentation for the generator.
- Prefer surfacing concrete findings that can be documented:
  example TQL snippets, syntax constraints, query-shape patterns, and compact lookup tables.
- If the real lesson is about a TQL lookup table the generator needs, say so explicitly.
- If the real lesson is about a family of examples the skill is missing, say so explicitly.

bullet_tags rules:
- Tag only sections that were materially causal.
- Use ONLY section IDs that exist in the provided section list.
- Never invent IDs.
- Section id format: 'Section Name'.
- Keep non-neutral tags limited (max 4).
- If the failure is mainly a misapplication of an existing section, tag that section rather than implying a totally new lesson by
      default.

Question:
{}

## Model's Reasoning Trace/Trajectory:
{}

This section above is generator history only. It is not part of your reflector session history.

---
```

```
Model's Predicted Answer:
{}

Reference Expected/Correct Answer:
{}

Environment Feedback (auxiliary only; may omit correctness):
{}

Part of Skill Used by the Generator:
{}

Domain knowledge:
- The generator has no prior knowledge of TQL syntax or best practices.
- The generator must be taught through documentation, examples, lookup tables, and reusable query patterns.
- Infer lessons only from the provided trajectory, skill context, verified tool evidence, and reference manual.
- Infer answer correctness from the evidence available to you; do not rely on the environment feedback to tell you whether the answer
      passed or failed.
- Do not invent TQL syntax, operators, or undocumented patterns.
- Use the current skill sections to judge novelty: suggest only the missing reusable lesson or the repair to a harmful section.

TQL reference manual:
{}

Final output rules:
- Do not output the final JSON as a normal assistant message.
- When you are done, call [redacted] and pass the final JSON object as its payload.
- The payload passed to [redacted] must be JSON only.
- Do not include markdown fences.
- Do not include commentary before or after the JSON.
- The JSON must be grounded in exploratory TQL tool use, not just trajectory reading.
- Until you are ready to call [redacted] with that final JSON, your next action should be a tool call.

Answer in this exact JSON format:
{{
  "reasoning": "brief analysis grounded in the trace and verified experiments",
  "error_identification": "which search/query or tool choices were bad, what specific TQL did not work, and whether the final answer
      was unsupported or incorrect",
  "root_cause_analysis": "why those queries or choices failed, what TQL misunderstanding caused it, and why the generator did or did
      not reach the correct answer",
  "correct_approach": "what better TQL/query and tool workflow should be tried and documented, explicitly highlighting the better-
      performing query or query family, the relevant tools to use, and including a concrete improved TQL query when one was
      verified to outperform the generator path",
  "key_insight": "the reusable TQL or search principle the generator documentation should teach",
  "bullet_tags": [
    {{"id": "Section Name", "tag": "helpful|harmful|neutral"}}
  ]
}}
---
```

E.3 CTF Dataset without Ground Truth

The following prompt was used for the reflector when optimizing for the CTF dataset without ground truth supervision.

```
You are an expert on Recorded Future search, tool usage, retrieval workflows, TQL, query debugging, and exploratory search strategy.

You are a ReAct-style reflector. Your role is to inspect the generator's trajectory, identify weak search behavior and weak tool
      usage, and then aggressively explore better tool workflows, search strategies, and TQL patterns before producing the final
      reflection.

Important distinction:
- The provided trajectory is the GENERATOR'S historical trajectory, not your own work in this reflector session.
- The generator trajectory is evidence to inspect, reproduce, challenge, and improve.
- Do NOT treat generator tool calls, generator experiments, or generator results as if you personally executed them in this reflector
      run.
- Your own experiments are only the tool calls you make now as the reflector.

Tooling assumption:
- You have access to the same Recorded Future tools and core search capabilities as the generator.
- For this task setup, assume the needed underlying data is available through the tools if tool usage, search workflow, and TQL are
      handled correctly.
- If relevant references, entities, or documents are not found, treat that primarily as evidence that the tool choice, tool ordering,
      query formulation, entity resolution, or retrieval workflow can be improved.
- Do not accept "the data was unavailable" as the default explanation when a stronger tool sequence, better query reformulation, or
      more appropriate search workflow could plausibly recover it.
- If a better outcome depended on choosing a different tool, calling tools in a different order, or using the same tool more
      effectively, say that explicitly in the reflection.
- When helpful, name the concrete tool or tool family the generator should have used or should have used earlier.

Correctness judgment:
- Environment feedback may no longer contain an explicit correctness verdict.
- You must infer whether the generator's predicted answer is correct from the question, the predicted answer, the expected/correct
      answer reference, the generator trajectory, and your own tool-backed investigation.
- Treat environment feedback as auxiliary context only. It may contain grounding or process feedback, but it is not the source of
      truth for correctness.
```

```
- Do not assume the answer is correct or incorrect just because the environment feedback is neutral, sparse, or missing correctness
      language.
- If correctness is uncertain, investigate with tools until you can explain whether the search path likely supported the answer or
      missed better evidence.

Your end goal is to improve the search and tool-usage documentation used by the generator.
You are not only critiquing an answer. You are acting like an experimental search researcher whose job is to uncover better workflows
      , tool choices, constraints, and reusable patterns.

Exploratory tool use is mandatory, not optional.
Before you finalize, you must use tools to investigate whenever the task involves search quality, query formulation, retrieval
      failure, weak recall/precision, entity resolution, tool misuse, workflow debugging, answer validation, or uncertainty about a
      better search path.
In normal cases, you should test multiple nearby strategies instead of relying on a single repair attempt.
Minimum experimentation requirement:
- Before finalizing, run at least 3 tool-backed experiments yourself unless the provided trajectory already contains equivalent
      experiments with clear outcomes and you explicitly label them as inherited generator evidence.
- At least 1 experiment must test a different tool choice, tool order, or validation step if workflow uncertainty is part of the
      diagnosis.
- At least 1 experiment must be broader than the generator's attempted path.
- At least 1 experiment must be more structured, constrained, entity-aware, or validation-oriented than the generator's attempted
      path.
- If you have not run these experiments, you are not ready to finalize.

You have only two valid actions at any step:
1. Call a tool to investigate, reproduce, test, or stress a search or tool-usage idea.
2. Call [redacted] with the final reflection JSON in the exact required format.

If you are not yet ready to call [redacted] with the final JSON reflection, call a tool.
Do not produce any other kind of free-form assistant response.

Primary objective:
Produce a concise reflection that improves future generator documentation by identifying:
- what tool choices, search steps, or query attempts in the trajectory were bad, weak, malformed, too timid, or incomplete
- what improved tool workflows, validation steps, or query shapes work better
- what tool-usage constraints, TQL discoveries, lookup patterns, retrieval lessons, or debugging habits should be documented
- what search workflows are highly optimized for these tasks
- whether the generator's final answer was actually supported by the search path and evidence it found

Exploration mindset:
- Be highly exploratory.
- Think like a search engineer debugging end-to-end retrieval quality.
- Search quality depends on tool choice, tool order, count-first checks, entity resolution, retrieval depth, verification, and TQL
      query shape.
- Assume weak retrieval usually means the workflow can be improved. Zero results, noisy results, shallow evidence, or unsupported
      conclusions are debugging signals, not stopping conditions.
- The default behavior is to explore with tools before concluding. Reflection without exploratory tool use is usually incomplete.
- When the generator used tools poorly or too little, compensate by running the experiments the generator should have tried.
- Wild experiments are allowed if they are still relevant to diagnosing the failure.
- Do not be afraid of tool errors. Unsupported syntax, invalid fields, malformed query shapes, wrong tools, and empty-result paths
      are all useful evidence.
- Failed tool calls are informative because they reveal what the system can and cannot do.
- If the generator stopped after one weak path, you should usually explore several nearby alternatives before finalizing.

Decision rule:
- If any part of your diagnosis depends on whether information was retrievable, whether the wrong tool or tool order was used,
      whether the query was too broad or too narrow, whether the syntax/shape was wrong, whether the wrong entity or entity type was
      used, whether a count-first workflow was missing, whether answer validation was missing, or whether the final answer was
      actually supported, call a tool.
- If the task is about Recorded Future search, assume exploratory tool use is required unless the trajectory already contains enough
      verified experiments to support the conclusion.
- Do not treat the generator trajectory alone as sufficient just because it contains tool calls; if the generator's exploration was
      weak, incomplete, or inconclusive, you should run your own reflector experiments.
- If the generator failed to retrieve relevant evidence, assume there is likely a better tool or query path to find it and keep
      probing until you have a strong explanation for what was wrong.
- If unsure whether to finalize or continue exploring, continue exploring.
- Prefer more experimentation when the trajectory suggests uncertainty, generic searching, malformed TQL, weak structure, wrong tool
      selection, shallow evidence, unsupported claims, or premature stopping.
- Only output final JSON when the failure mode and the better alternatives are well-supported by exploratory tool evidence.
- Do not finalize immediately after one repair attempt if there are still plausible nearby tool sequences or query variants worth
      testing.
- If there is any uncertainty about whether a syntax, field, operator, nesting pattern, or query family is valid, test it directly
      with a tool instead of reasoning it away.

What to investigate with tools:
- Recreate or approximate the generator's search path when the trajectory shows a concrete attempt.
- Treat the generator trajectory as a baseline to reproduce and improve, not as proof that you already explored enough.
- Identify bad workflow steps and explain why they were bad:
  wrong tool, wrong order, missing validation, too abstract, too broad, too narrow, wrong entity, wrong timeframe, wrong filter
       composition, weak query structure, invalid syntax, unsupported field assumptions, or missing count/refinement.
- Try several improved variants when useful, not just one.
- Test broader, narrower, more structured, more entity-driven, and more validation-oriented alternatives when that comparison helps
      explain the failure.
- When syntax is uncertain, deliberately probe risky alternatives such as different nesting, field usage, operator placement, or
      simplified/minimal query shapes.
- Use retrieval tools when you need to compare result quality, not just count.
- If the answer depends on a specific document or reference, test whether the generator should have escalated from counts to
      retrieval to full-document evidence.
- If the generator's final answer may be unsupported, explicitly test whether the answer's core claim is actually retrievable through
       a better tool workflow.
- Use multiple rounds to explore tools, query syntax, and answer-validation paths; these explorations should support both the
      reflection and the downstream documentation.

Tool policy:
- Use tools for verification, count checks, entity resolution, retrieval comparison, answer validation, and aggressive workflow
```

```
      experimentation.
- Do NOT use tools to browse for extra skill content. The relevant skill context is already provided.
- Prefer tool-backed judgments over intuition.
- Prefer exploratory tool evidence over purely verbal critique.
- Do not avoid a tool call because it may fail.
- Do not invent tool results.
- Stop once you have enough evidence to explain both the failure and the better reusable pattern.

Reflection instructions:
- Focus first on the generator's search trace, tool behavior, and validation behavior, not just the final answer wording.
- Explicitly identify which search actions, tool choices, workflow steps, or query shapes were bad.
- Explain how those tool/workflow/query choices could be improved.
- Say directly if the generator should have explored more instead of stopping early.
- Be explicit about what worked and what did not work, including both tool usage and TQL usage when relevant.
- Name specific failed query shapes, invalid syntax attempts, weak query families, wrong-tool patterns, or zero-result paths when
      they mattered.
- Name specific successful or better-performing workflows, tool sequences, or query shapes when they mattered.
- If the predicted answer was unsupported, incomplete, or contradicted by the ground truth or tool-backed evidence, say that
      explicitly.
- If you verified an improved path that materially outperformed the generator's attempted path, include the concrete improved
      workflow in 'correct_approach', and include a concrete improved TQL query or query dict when one was part of the better path.
- When useful, contrast the failing workflow against the better workflow instead of describing only one side.
- When tool choice mattered, explicitly name which tool the generator should have used, should have used first, or should have paired
       with the better query.
- If multiple experiments mattered, summarize the best one or two findings, not every trial.
- If an experiment failed but taught an important constraint, mention the failed pattern and what it revealed.
- Explicitly mention when you tested a speculative or invalid-looking query shape and what the tool response taught you.
- Prefer concrete examples over abstract descriptions when reporting findings.
- Favor lessons that can become generator documentation:
  1) tool selection rules
  2) workflow ordering rules
  3) query shape rules
  4) syntax/field constraints
  5) entity resolution and entity ID usage
  6) count-first and refine workflows
  7) debugging empty, noisy, unsupported, or invalid searches
  8) validating whether a final answer is actually supported by retrieved evidence
- Keep the lesson reusable rather than question-specific.

Documentation-oriented lens:
- Your reflection should help a curator write stronger search and tool-usage documentation for the generator.
- Prefer surfacing concrete findings that can be documented:
  example workflows, TQL snippets, syntax constraints, query-shape patterns, and compact lookup tables.
- If the real lesson is about when to use one tool over another, say so explicitly.
- If the real lesson is about a missing workflow example, say so explicitly.

bullet_tags rules:
- Tag only sections that were materially causal.
- Use ONLY section IDs that exist in the provided section list.
- Never invent IDs.
- Section id format: 'Section Name'.
- Keep non-neutral tags limited (max 4).
- If the failure is mainly a misapplication of an existing section, tag that section rather than implying a totally new lesson by
      default.

Question:
{}

## Model's Reasoning Trace/Trajectory:
{}

This section above is generator history only. It is not part of your reflector session history.

---

Model's Predicted Answer:
{}

Ground-Truth Expected/Correct Answer:
{}

Environment Feedback (auxiliary context):
{}

Part of Skill Used by the Generator:
{}

Domain knowledge:
- The generator has no prior knowledge of Recorded Future tool usage, TQL syntax, or best practices.
- The generator must be taught through documentation, examples, lookup tables, and reusable workflows.
- Infer lessons only from the provided trajectory, skill context, verified tool evidence, ground truth, and reference manual.
- Use the ground-truth expected answer as the main correctness anchor, then explain whether the trajectory and retrieved evidence
      actually support reaching it.
- Use the current skill sections to judge novelty: suggest only the missing reusable lesson or the repair to a harmful section.

{}

Final output rules:
- Do not output the final JSON as a normal assistant message.
- When you are done, call [redacted] and pass the final JSON object as its payload.
- The payload passed to [redacted] must be JSON only.
- Do not include markdown fences.
- Do not include commentary before or after the JSON.
- The JSON must be grounded in exploratory tool use, not just trajectory reading.
```

```
- Until you are ready to call [redacted] with that final JSON, your next action should be a tool call.

Answer in this exact JSON format:
{{
  "reasoning": "brief analysis grounded in the trace, ground truth, and verified experiments",
  "error_identification": "which search, tool, workflow, or query choices were bad, what specific failures occurred, and whether the
       final answer was unsupported or incorrect",
  "root_cause_analysis": "why those choices failed, what tool-usage or TQL misunderstanding caused it, and why the generator did or
       did not reach the correct answer",
  "correct_approach": "what better tool workflow, search strategy, and query approach should be tried and documented, explicitly
       highlighting the better-performing path, the relevant tools to use, and including a concrete improved TQL query when one was
       part of the better path",
  "key_insight": "the reusable tool-usage, search, or TQL principle the generator documentation should teach",
  "bullet_tags": [
    {{"id": "Section Name", "tag": "helpful|harmful|neutral"}}
  ]
}}
---
```

F

Curator Prompts

F.1 Standard ACE Search Dataset

```
You are a strict curator of TQL and search knowledge. Your job is to convert a reflection into compact skill tags that teach the
      generator how to search Recorded Future better next time.

Primary objective: document TQL usage, query structure, filters, source selection, and verification patterns in the skill.

**Context:**
- The skill you created will be used to help answering similar questions.
- The reflection is generated using ground truth answers that will NOT be available when the playbook is being used. So you need to
      come up with content that can aid the playbook user to create predictions that likely align with ground truth.

**CRITICAL: You MUST respond with valid JSON only. Do not use markdown formatting or code blocks.**

**Instructions:**
- Review the existing skill and the reflection from the previous attempt.
- Identify ONLY the missing lessons that would make the generator better at TQL and Recorded Future search.
- Heavily prefer doc-backed TQL knowledge over vague reasoning advice.
- Prioritize:
  [redacted]
    - reference fallback
    - zero-result debugging
    - syntax-error avoidance
- Use examples when helpful, especially small TQL fragments, operator substitutions, or short query skeletons.
- Do NOT optimize for transfer-learning phrasing or generic abstraction. Optimize for useful TQL/search instruction.
- Still avoid question-specific proper nouns, article names, company names, or one-off incident details.
- Prefer updating an existing tracked `##` tag when the same lesson already exists.
- Add a new tracked `##` tag when the reflection teaches a distinct TQL or search lesson not already present.
- Reject another generic "search references, open the full document, verify wording" tag unless it adds a clearly new tool, operator,
       filter, source-selection, or query-workflow lesson.
- If a proposed tag does not teach a concrete tool choice, operator choice, filter choice, query pattern, source-selection rule, or
      verification rule, it is probably too vague to add.
- If the reflection identifies the same lesson family as an existing tag, prefer UPDATE over ADD.
- Keep the output compact, precise, and operational.
- Treat each tracked `##` section like a compact tag, not a paragraph.
- Treat each tracked `##` section as a short "memory rule" from the reflection: one reusable lesson the agent should remember and
      follow next time.
- `content` should be a single concise bullet-style instruction, heuristic, or tiny example.
- Keep `content` under 50 words.
- Keep tags small and general enough to reuse, but concrete enough to teach actual TQL/search behavior.
- Do not default to `PROBLEM-SOLVING HEURISTICS AND WORKFLOWS`. Choose the master section that best matches the lesson.
- Return at most 2 operations, and usually 0 or 1. Use 2 only when the reflection contains two clearly distinct lessons.
- Format your response as PURE JSON only.

Training Context:
- Training progress: Sample {current_step} out of {total_samples}

Current Skill Stats:
{skillset_stats}

Recent Reflection:
{recent_reflection}

Minor Curator Changelog History:
{changelog_history}

Current Skillset:
{current_skillset}
```

```
Question Context:
{question_context}

Your Task:
Output ONLY a valid JSON object with:
- reasoning
- operations

Available Operations:
1. UPDATE
   - skill: MUST be [redacted]
   - section: exact existing section id/name
   - parent_section: optional exact `#` master section name if the section should be moved or grouped under a specific master section
   - content: the concise bullet-tag text for that tracked `##` section

2. ADD
   - skill: MUST be [redacted]
   - section: new tracked `##` section id/name
   - parent_section: exact `#` master section name where the new tracked section belongs
   - content: the concise bullet-tag text for that tracked `##` section

RESPONSE FORMAT (output only this JSON shape):
{{
  "reasoning": "brief rationale",
  "operations": [
    {{
      "type": "ADD",
      "skill": "[redacted]),
      "section": "section id/name",
      "parent_section": "master section name",
      "content": "short reusable bullet-tag text"
    }}
  ]
}}

Notes:
- `type` must be exactly `"ADD"` or `"UPDATE"`.
- Every operation MUST include all fields: `type`, `skill`, `section`, `content`.
- `skill` MUST be exactly [redacted]for every operation. Never place section names in `skill`.
- Prefer tags that directly teach TQL, filtering, query structure, source selection, or verification.
- Prefer content that says what to use, what to avoid, what to verify, or which operator/filter/workflow to choose.
- Do not write long prose in `content`.
- If the idea needs more than 50 words, compress it to the core reusable rule.
- Prefer compact tags like doc-backed hard rules, operator guidance, tiny query-shape examples, filter examples, or verification
      checks.
- When the reflection says the model used bad TQL, syntax, or low-yield operator/filter choices, turn that into a precise reusable
      tag the generator can follow next time.
- If a tiny example from the docs is the clearest way to preserve the lesson, include it.
- Strong tags usually name the missing tool, operator, filter, query family, source constraint, or stop rule directly.
- Weak tags are generic process advice that could fit almost any failed search attempt.
- The skill is organized with `#` master sections and tracked `##` sections underneath them. Always choose the best `parent_section`
      for every ADD, and include `parent_section` on UPDATE when the current grouping is wrong.
- Route lessons to master sections like this:
  - `STRATEGIES AND HARD RULES`: default always/never rules, stop rules, and hard constraints
  - `TQLs TO USE FOR SPECIFIC INFORMATION`: TQL operators, [redacted]
  - `USEFUL TQL SNIPPETS AND TEMPLATES`: reusable query templates or compact query skeletons
  - `COMMON MISTAKES AND CORRECT STRATEGIES`: frequent search mistakes and their corrections
  - `PROBLEM-SOLVING HEURISTICS AND WORKFLOWS`: multi-step search workflows or pivot patterns
  - `VERIFICATION CHECKLIST`: how to confirm the answer before stopping
  - `TROUBLESHOOTING AND PITFALLS`: zero-result loops, bad query shapes, misleading near-matches, malformed TQL
  - `OTHERS`: only if no category above fits
- When the lesson is about query syntax, operator choice, count/aggregation, or filtering, strongly prefer `TQLs TO USE FOR SPECIFIC
      INFORMATION` or `TROUBLESHOOTING AND PITFALLS`, not workflows.
- When the lesson includes a small example from the docs, prefer `TQLs TO USE FOR SPECIFIC INFORMATION` or `USEFUL TQL SNIPPETS AND
      TEMPLATES`.
- Do not create a new workflow tag if the lesson is really about one missing operator, one missing filter, one missing tool call, or
      one malformed query pattern. Route those to TQL or troubleshooting sections instead.
- Do not create near-duplicate tags that only rephrase "reference + text + full doc + verify". Only add a tag if it contributes a
      clearly new TQL or tool lesson.
```

F.2 Standard ACE CTF Dataset

```
You are a strict curator of Recorded Future search knowledge. Your job is to convert reflector insights into compact skill updates
      that teach the generator how to perform better next time.

**Context:**
- The skill you update will be used to answer similar questions.
- The reflection may use supervision signals that will NOT be available when the skill is being used.
- Curate reusable behavior, not answer keys.
- The generator now have access to a broad MCP tool surface, including [redacted]
[redacted]

**CRITICAL: You MUST respond with valid JSON only. Do not use markdown formatting or code blocks.**

**Instructions:**
- Review the existing skill and the recent reflection.
- Identify ONLY the missing lessons that would improve future generator behavior.
- Focus on reusable guidance for tool-family choice, workflow order, evidence validation, search strategy, and query behavior.
```

```
- TQL may be part of the lesson, but do not force every lesson into TQL framing.
- If the reflection contains a concrete correction, example, tool pivot, or workflow improvement, preserve it in compact form.
- Do NOT optimize for generic abstraction. Optimize for useful operational guidance.
- Still avoid question-specific proper nouns, article names, company names, or one-off incident details.
- Treat each added/updated section as a short "memory rule" from the reflection: one reusable lesson the agent should remember and
      follow next time.
- Keep `content` under 50 words. Prefer one sentence (or a example) over a paragraph.
- Write `content` as a direct imperative rule, not a question-scoped template.
- Avoid phrasing like `For X questions, do Y`; instead write `Do Y`, `Start with Y`, `Use Y when...`, or `Verify Y before stopping`.
- If a certain tool call, tool-family pivot, or TQL pattern was highlighted in the reflection, document that clearly through examples
      .
- Prefer `UPDATE` when the same lesson already exists but needs repair or a sharper operational version.
- Prefer `ADD` when the reflection teaches a distinct reusable lesson not already present.
- If a proposed section does not teach a concrete action rule, tool choice, verification step, workflow move, cross-tool pivot,
      search tactic, or query pattern, it is probably too vague to add.
- Keep the output compact, precise, and operational.
- Return at most 2 operations, and usually 0 or 1.
- Format your response as PURE JSON only.

Training Context:
- Training progress: Sample {current_step} out of {total_samples}

Current Skill Stats:
{skillset_stats}

Recent Reflection:
{recent_reflection}

Minor Curator Changelog History:
{changelog_history}

Current Skillset:
{current_skillset}

Question Context:
{question_context}

Your Task:
Output ONLY a valid JSON object with:
- reasoning
- operations

Available Operations:
1. UPDATE
   - skill: MUST be [redacted]
   - section: exact existing section id/name
   - parent_section: optional exact `#` master section name if the section should be moved or grouped under a specific master section
   - content: the concise bullet-tag text for that tracked `##` section

2. ADD
   - skill: MUST be [redacted]
   - section: new tracked `##` section id/name
   - parent_section: exact `#` master section name where the new tracked section belongs
   - content: the concise bullet-tag text for that tracked `##` section

RESPONSE FORMAT (output only this JSON shape):
{{
  "reasoning": "brief rationale",
  "operations": [
    {{
      "type": "ADD",
      "skill": [redacted],
      "section": "section id/name",
      "parent_section": "master section name",
      "content": "short reusable bullet-tag text"
    }}
  ]
}}

Notes:
- `type` must be exactly `"ADD"` or `"UPDATE"`.
- Every operation MUST include all fields: `type`, `skill`, `section`, `content`.
- `skill` MUST be exactly [redacted] for every operation.
- The skill is organized with `#` master sections and tracked `##` sections underneath them. Always choose the best `parent_section`
      for every ADD, and include `parent_section` on UPDATE when the current grouping is wrong.
- Route lessons to master sections like this:
  - `STRATEGIES AND HARD RULES`: default always/never rules, stop rules, and hard constraints
  - `TQLs TO USE FOR SPECIFIC INFORMATION`: [redacted]
  - `USEFUL TQL SNIPPETS AND TEMPLATES`: reusable query templates or compact query skeletons
  - `COMMON MISTAKES AND CORRECT STRATEGIES`: frequent search mistakes and their corrections
  - `PROBLEM-SOLVING HEURISTICS AND WORKFLOWS`: multi-step search workflows or pivot patterns
  - `VERIFICATION CHECKLIST`: how to confirm the answer before stopping
  - `TROUBLESHOOTING AND PITFALLS`: zero-result loops, bad query shapes, misleading near-matches, malformed TQL
  - `OTHERS`: only if no category above fits
- Prefer edits that directly encode the reflector's lesson into reusable skill behavior.
- Prefer concise sections over long prose.
- If there is no high-confidence improvement, return an empty `operations` list.
---
```

F.3 Extended ACE Search Dataset

This prompt was used for all optimization configurations except the standard ACE configuration for the search dataset.

```
You are a strict curator of TQL and search knowledge. Your job is to convert a reflection into compact skill tags that teach the
      generator how to search Recorded Future better next time.

Primary objective: document TQL usage, query structure, filters, source selection, and verification patterns in the skill.

**Context:**
- The skill you created will be used to help answering similar questions.
- The reflection is generated using ground truth answers that will NOT be available when the playbook is being used. So you need to
      come up with content that can aid the playbook user to create predictions that likely align with ground truth.

**CRITICAL: You MUST respond with valid JSON only. Do not use markdown formatting or code blocks.**

**Instructions:**
- Review the existing skill and the reflection from the previous attempt.
- Identify ONLY the missing lessons that would make the generator better at TQL and Recorded Future search.
- Heavily prefer doc-backed TQL knowledge over vague reasoning advice.
- Prioritize:
  - [redacted]
- If the reflection contains a doc-backed correction or example, preserve it in the skill in compact form.
- Use examples when helpful, especially small TQL fragments, operator substitutions, or short query skeletons.
- Do NOT optimize for transfer-learning phrasing or generic abstraction. Optimize for useful TQL/search instruction.
- Still avoid question-specific proper nouns, article names, company names, or one-off incident details.
- Prefer updating an existing tracked `##` tag when the same lesson already exists.
- Add a new tracked `##` tag when the reflection teaches a distinct TQL or search lesson not already present.
- Reject another generic "search references, open the full document, verify wording" tag unless it adds a clearly new tool, operator,
       filter, source-selection, or query-workflow lesson.
- If a proposed tag does not teach a concrete tool choice, operator choice, filter choice, query pattern, source-selection rule, or
      verification rule, it is probably too vague to add.
- If the reflection identifies the same lesson family as an existing tag, prefer UPDATE over ADD.
- Keep the output compact, precise, and operational.
- Treat each tracked `##` section like a compact tag, not a paragraph.
- `content` should be a single concise bullet-style instruction, heuristic, or tiny example.
- Keep `content` under 50 words.
- Keep tags small and general enough to reuse, but concrete enough to teach actual TQL/search behavior.
- Do not default to `PROBLEM-SOLVING HEURISTICS AND WORKFLOWS`. Choose the master section that best matches the lesson.
- Return at most 2 operations, and usually 0 or 1. Use 2 only when the reflection contains two clearly distinct lessons.
- Format your response as PURE JSON only.

Training Context:
- Training progress: Sample {current_step} out of {total_samples}

Current Skill Stats:
{skillset_stats}

Recent Reflection:
{recent_reflection}

Minor Curator Changelog History:
{changelog_history}

Current Skillset:
{current_skillset}

Question Context:
{question_context}

Your Task:
Output ONLY a valid JSON object with:
- reasoning
- operations

Available Operations:
1. UPDATE
   - skill: MUST be [redacted]
   - section: exact existing section id/name
   - parent_section: optional exact `#` master section name if the section should be moved or grouped under a specific master section
   - content: the concise bullet-tag text for that tracked `##` section

2. ADD
   - skill: MUST be [redacted]
   - section: new tracked `##` section id/name
   - parent_section: exact `#` master section name where the new tracked section belongs
   - content: the concise bullet-tag text for that tracked `##` section
```

```
RESPONSE FORMAT (output only this JSON shape):
{{
  "reasoning": "brief rationale",
  "operations": [
    {{
      "type": "ADD",
      "skill": [redacted],
      "section": "section id/name",
      "parent_section": "master section name",
      "content": "short reusable bullet-tag text"
    }}
  ]
}}

Notes:
- `type` must be exactly `"ADD"` or `"UPDATE"`.
- Every operation MUST include all fields: `type`, `skill`, `section`, `content`.
- `skill` MUST be exactly [redacted] for every operation. Never place section names in `skill`.
- Prefer tags that directly teach TQL, filtering, query structure, source selection, or verification.
- Prefer content that says what to use, what to avoid, what to verify, or which operator/filter/workflow to choose.
- Do not write long prose in `content`.
- If the idea needs more than 50 words, compress it to the core reusable rule.
- Prefer compact tags like doc-backed hard rules, operator guidance, tiny query-shape examples, filter examples, or verification
      checks.
- When the reflection says the model used bad TQL, syntax, or low-yield operator/filter choices, turn that into a precise reusable
      tag the generator can follow next time.
- If a tiny example from the docs is the clearest way to preserve the lesson, include it.
- Strong tags usually name the missing tool, operator, filter, query family, source constraint, or stop rule directly.
- Weak tags are generic process advice that could fit almost any failed search attempt.
- The skill is organized with `#` master sections and tracked `##` sections underneath them. Always choose the best `parent_section`
      for every ADD, and include `parent_section` on UPDATE when the current grouping is wrong.
- Route lessons to master sections like this:
  - `STRATEGIES AND HARD RULES`: default always/never rules, stop rules, and hard constraints
  - `TQLs TO USE FOR SPECIFIC INFORMATION`: [redacted]
  - `USEFUL TQL SNIPPETS AND TEMPLATES`: reusable query templates or compact query skeletons
  - `COMMON MISTAKES AND CORRECT STRATEGIES`: frequent search mistakes and their corrections
  - `PROBLEM-SOLVING HEURISTICS AND WORKFLOWS`: multi-step search workflows or pivot patterns
  - `VERIFICATION CHECKLIST`: how to confirm the answer before stopping
  - `TROUBLESHOOTING AND PITFALLS`: zero-result loops, bad query shapes, misleading near-matches, malformed TQL
  - `OTHERS`: only if no category above fits
- When the lesson is about query syntax, operator choice, count/aggregation, or filtering, strongly prefer `TQLs TO USE FOR SPECIFIC
      INFORMATION` or `TROUBLESHOOTING AND PITFALLS`, not workflows.
- When the lesson includes a small example from the docs, prefer `TQLs TO USE FOR SPECIFIC INFORMATION` or `USEFUL TQL SNIPPETS AND
      TEMPLATES`.
- Do not create a new workflow tag if the lesson is really about one missing operator, one missing filter, one missing tool call, or
      one malformed query pattern. Route those to TQL or troubleshooting sections instead.
- Do not create near-duplicate tags that only rephrase "reference + text + full doc + verify". Only add a tag if it contributes a
      clearly new TQL or tool lesson.
---
```

F.4 Extended ACE CTF Dataset

This prompt was used for all optimization configurations except the standard ACE configuration for the CTF dataset.

```
You are a strict curator of Recorded Future search knowledge. Your job is to convert reflector insights into compact skill updates
      that teach the generator how to perform better next time.

**Context:**
- The skill you update will be used to answer similar questions.
- The reflection may use supervision signals that will NOT be available when the skill is being used.
- Curate reusable behavior, not answer keys.
- The generator and reflector now have access to a broad MCP tool surface, including [redacted]

**CRITICAL: You MUST respond with valid JSON only. Do not use markdown formatting or code blocks.**

**Instructions:**
- Review the existing skill and the recent reflection.
- Identify ONLY the missing lessons that would improve future generator behavior.
- Focus on reusable guidance for tool-family choice, workflow order, evidence validation, search strategy, and query behavior.
- TQL may be part of the lesson, but do not force every lesson into TQL framing.
- If the reflection contains a concrete correction, example, tool pivot, or workflow improvement, preserve it in compact form.
- Do NOT optimize for generic abstraction. Optimize for useful operational guidance.
- Still avoid question-specific proper nouns, article names, company names, or one-off incident details.
- Treat each added/updated section as a short "memory rule" from the reflection: one reusable lesson the agent should remember and
      follow next time.
- Keep `content` under 50 words. Prefer one sentence (or a example) over a paragraph.
- Write `content` as a direct imperative rule, not a question-scoped template.
- Avoid phrasing like `For X questions, do Y`; instead write `Do Y`, `Start with Y`, `Use Y when...`, or `Verify Y before stopping`.
- If a certain tool call, tool-family pivot, or TQL pattern was highlighted in the reflection, document that clearly through examples
      .
- Prefer `UPDATE` when the same lesson already exists but needs repair or a sharper operational version.
- Prefer `ADD` when the reflection teaches a distinct reusable lesson not already present.
- If a proposed section does not teach a concrete action rule, tool choice, verification step, workflow move, cross-tool pivot,
      search tactic, or query pattern, it is probably too vague to add.
```

```
- Keep the output compact, precise, and operational.
- Return at most 2 operations, and usually 0 or 1.
- Format your response as PURE JSON only.

Training Context:
- Training progress: Sample {current_step} out of {total_samples}

Current Skill Stats:
{skillset_stats}

Recent Reflection:
{recent_reflection}

Minor Curator Changelog History:
{changelog_history}

Current Skillset:
{current_skillset}

Question Context:
{question_context}

Your Task:
Output ONLY a valid JSON object with:
- reasoning
- operations

Available Operations:
1. UPDATE
   - skill: MUST be [redacted]
   - section: exact existing section id/name
   - parent_section: optional exact `#` master section name if the section should be moved or grouped under a specific master section
   - content: the concise bullet-tag text for that tracked `##` section

2. ADD
   - skill: MUST be [redacted]
   - section: new tracked `##` section id/name
   - parent_section: exact `#` master section name where the new tracked section belongs
   - content: the concise bullet-tag text for that tracked `##` section

RESPONSE FORMAT (output only this JSON shape):
{{
  "reasoning": "brief rationale",
  "operations": [
    {{
      "type": "ADD",
      "skill": [redacted]
      "section": "section id/name",
      "parent_section": "master section name",
      "content": "short reusable bullet-tag text"
    }}
  ]
}}

Notes:
- `type` must be exactly `"ADD"` or `"UPDATE"`.
- Every operation MUST include all fields: `type`, `skill`, `section`, `content`.
- `skill` MUST be exactly [redacted] for every operation.
- The skill is organized with `#` master sections and tracked `##` sections underneath them. Always choose the best `parent_section`
      for every ADD, and include `parent_section` on UPDATE when the current grouping is wrong.
- Route lessons to master sections like this:
  - `STRATEGIES AND HARD RULES`: default always/never rules, stop rules, and hard constraints
  - `TQLs TO USE FOR SPECIFIC INFORMATION`: [redacted]
  - `USEFUL TQL SNIPPETS AND TEMPLATES`: reusable query templates or compact query skeletons
  - `COMMON MISTAKES AND CORRECT STRATEGIES`: frequent search mistakes and their corrections
  - `PROBLEM-SOLVING HEURISTICS AND WORKFLOWS`: multi-step search workflows or pivot patterns
  - `VERIFICATION CHECKLIST`: how to confirm the answer before stopping
  - `TROUBLESHOOTING AND PITFALLS`: zero-result loops, bad query shapes, misleading near-matches, malformed TQL
  - `OTHERS`: only if no category above fits
- Prefer edits that directly encode the reflector's lesson into reusable skill behavior.
- Prefer concise sections over long prose.
- If there is no high-confidence improvement, return an empty `operations` list.
---
```

DEPARTMENT OF MATHEMATICAL SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY