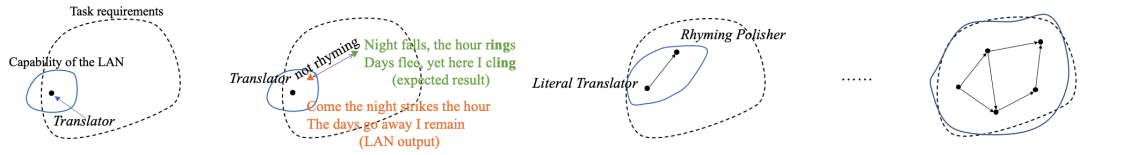
# A Human-Computer Collaborative Tool for Training a Single Large Language Model Agent into a Network through Few Examples

LIHANG PAN*, Department of Computer science and Technology, Tsinghua University, China
YUXUAN LI*, Department of Computer science and Technology, Tsinghua University, China
CHUN YU†, Department of Computer science and Technology, Tsinghua University, China
YUANCHUN SHI, Department of Computer science and Technology, Tsinghua University, China




(1) Only one agent (*Translator*) in the initial network
(2) The result for the example input is not satisfactory
(3) The *Translator* is split into two agents. (*Literal Translator* & *Rhyming Polisher*)
(n) With few examples, the initial agent evolves into a network

Fig. 1. How EasyLAN trains a task-oriented LLM agent network (LAN) from a single LLM agent. (1) EasyLAN auto-generates an initial LAN that only contains a single LLM agent based on the task (e.g., translating French to English). A significant gap exists between the capabilities of the initial LAN and the task requirements. (2) A training example consists of an input and a ground truth. For a given training example, EasyLAN identifies discrepancies between the LAN's output and the expected output. For instance, when the input is a line of French poetry, "Vienne la nuit sonne l'heure, les jours s'en vont je demeure", the LAN fails to translate the text accurately while preserving the original rhyming scheme. (3) EasyLAN identifies the cause of the discrepancies and updates the LAN with respect to both the network architecture (e.g., splitting *Translator* into *Literal Translator* and *Rhyming Polisher*) and agent contents (e.g., adjusting the functionality of an agent). (n) EasyLAN iterates over a small set of training examples and constructs a satisfactory LAN.

The capabilities of a single large language model (LLM) agent for solving a complex task are limited. Connecting multiple LLM agents to a network can effectively improve overall performance. However, building an LLM agent network (LAN) requires a substantial amount of time and effort. In this paper, we introduce EasyLAN, a human-computer collaborative tool that helps developers construct LANs. EasyLAN initially generates a LAN containing only one agent based on the description of the desired task. Subsequently, EasyLAN leverages a few training examples to update the LAN. For each example, EasyLAN models the gap between the output and the ground truth and identifies the causes of the errors. These errors are addressed through carefully designed strategies. Users can intervene in EasyLAN's workflow or directly modify the LAN. Eventually, the LAN evolves from a single agent to a network of LLM agents. The experimental results indicate that developers can rapidly construct LANs with good performance.

CCS Concepts: • **Human-centered computing** → **Interactive systems and tools**; **Ubiquitous and mobile computing systems and tools**.

Additional Key Words and Phrases: large language model, LLM agent network, human-computer collaboration

*Both authors contributed equally to the paper.
†Indicates the corresponding author.


Manuscript submitted to ACM

**ACM Reference Format:**
Lihang Pan, Yuxuan Li, Chun Yu, and Yuanchun Shi. 2024. A Human-Computer Collaborative Tool for Training a Single Large Language Model Agent into a Network through Few Examples. In *CHI '24: ACM CHI Conference on Human Factors in Computing Systems, May 11–16, 2024, Hawaii, USA.* ACM, New York, NY, USA, 30 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

The capabilities of Large Language Models (LLMs) in handling complex tasks are limited. A common solution is to decompose a complex task into sub-tasks [16, 29, 82], each managed by an LLM-driven agent. These agents connect and form a network, collaboratively accomplishing the complex task. However, existing LLM agent networks (LANs) require manual editing by developers [71], demanding significant time and effort in the network's design, testing, and modification. Developers must (1) estimate LLM capabilities to determine the task division and interconnections among agents [72] and (2) inspect each agent to identify and resolve issues that lead to suboptimal performance. Human-computer collaboration [34, 35] is a promising solution that leverages artificial intelligence (AI) to alleviate physical and mental burdens. Nevertheless, this collaborative paradigm has not been applied in the design and development of LANs.

In this paper, we introduce EasyLAN, a human-computer collaborative editing tool for developing task-oriented LLM agent networks (LANs). The most significant feature is the "few-example-driven" paradigm for LAN construction. EasyLAN alleviates the need for proactive decomposition of the complex task during LAN design and reduces the effort for reactive inspection and modification of agents during LAN debugging. As illustrated in Figure 1, EasyLAN initiates the LAN with a single agent based on a brief description of the complex task. For each training example, EasyLAN compares the actual and expected output to identify the limitations and the root causes within the LAN. EasyLAN then formulates and implements updates, improving the network's capabilities to accommodate the given example. The human-computer collaboration is manifested in the following way: the LAN developer supervises how EasyLAN executes the workflow mentioned above. When anomalies or errors occur, they can intervene and correct the system's actions to ensure appropriate execution.

To implement EasyLAN, we designed the internal structure of agents, which consists of input, control, execution, and output modules. The connection of input and output modules between different agents forms the network. The control module evaluates and decides whether the agent should be activated, while the execution module computes the agent output. Both of them leverage LLMs for computation and contain updatable knowledge bases and example repositories for few-shot learning. We designed strategies for updating the LAN and a pipeline for identifying error causes and selecting corresponding strategies. A user interface has been implemented in the browser to facilitate user inspection and control over the update process. The user can directly edit the LAN through GUI interactions (e.g., drag and drop) when necessary. We conducted a user study (N=12) to evaluate the usability of EasyLAN. The experimental results showed that EasyLAN can help users reduce interaction time by 39.3% while improving the performance of the constructed LANs by 39.8%.

In the remaining parts of this paper, we first review work related to our work. We then outline the design of the agent network (Section 3), which serves as the output of EasyLAN. Subsequently, we describe how the user and EasyLAN collaboratively train the LAN, detailing: 1) the automated mechanisms which EasyLAN employs to update the LAN (Section 4.1); and 2) how the user inspects and intervenes in the update process (Section 4.2). Technical details of the system implementation are elaborated upon in Section 5, focusing on the design of prompts, as the effectiveness of EasyLAN and the LAN heavily relies on LLMs. Finally, we validate the performance of EasyLAN through an evaluation study.

## 2 RELATED WORK

### 2.1 Decomposing Complex Tasks and Connecting Multiple LLM Agents

Large Language Models (LLMs) have been applied to a wide array of tasks such as code generation [39], storytelling [9], and command understanding [33]. However, their performance is limited in real-world tasks that are inherently complex and require domain-specific knowledge [10, 42] or multi-step reasoning [25, 51]. One promising approach to address these limitations is to explicitly employ multiple LLM-driven agents to collaborate on the tasks, simulating the human process of decomposing complex tasks and addressing them separately [4, 60, 78]. For instance, AI Chains [72] accomplish tasks like peer-review writing and personalized flashcard creation by chaining LLM prompts. In addition to performance gains, this approach has been proved to offer improved transparency and controllability [72] compared to embedding multi-step reasoning within a single LLM calculation (e.g., chain of thought [69]) or repetitive calculations with similar prompts (e.g., SayCan [5] and AutoCoT [80]).

There are two primary challenges in creating a network of LLM agents. First, the design of the network architecture, specifically the decomposition of complex tasks into subtasks, is not straightforward [32, 60, 72]. Users must consider global constraints [28, 76] and break down tasks into actionable subtasks [75]. These subtasks then need to be allocated to appropriate entities (e.g., self-sourcing [61], friend-sourcing [1, 44, 54], and crowd-sourcing [3]) based on their requirements such as expertise and task familiarity [24, 60]. The difficulty is further amplified when constructing an LLM network, as users struggle to predict LLM capabilities in solving specific problems [12]. Second, using existing graphical user interface (GUI) editing tools introduces additional interaction overhead. Although researchers have proposed GUI tools for editing LAN (e.g., PromptChainer [71]), users are confined to constructing, testing, and modifying the network through tedious GUI operations. Existing tools do not allow for human-machine collaboration to reduce users' workload.

EasyLAN addresses the aforementioned challenges in two ways. On the one hand, it obviates the need for users to design the network, thus reducing users' cognitive load. EasyLAN automatically tailors both the network structure and the content of each agent according to the provided training samples. On the other hand, EasyLAN automates the process of network update. Even if EasyLAN makes an error, users can intervene in the update pipeline and correct its behaviors through minimal interactions, thereby decreasing their physical workload.

### 2.2 Using Examples in LLM Applications

Existing research leverages few-shot learning [6, 52] to introduce examples (pairs of input and output) in the prompts. These examples guide the LLM to generate more accurate results. This method has substantially lowered the barriers to prototype AI applications, making it particularly beneficial for those without AI expertise. Consequently, this approach has been broadly adopted in various fields, including translation [52], chatbots [63], end-user programming [19], and robot interaction [46]. Although some studies have optimized the construction [14, 36, 58, 77], selection [2, 38, 53], and order [41, 81] of these examples to enhance model performance [15], they have not fundamentally changed how examples are employed within LLM-based applications.

One concern among researchers is what LLMs are capable of learning through few-shot learning [20, 43, 47, 68], a question critical to evaluating the method's reliability for real-world applications [56, 62, 74]. While preliminary insights and conclusions exist for AI experts ( e.g., few-shot examples may provide input distribution and output space [43]), there has been no comprehensive answer suitable for non-AI experts (e.g., human-computer interaction (HCI)

researchers without an AI background). This gap often forces these users into a 'trial-and-error' approach when creating prompts [12, 26], reducing efficiency, result quality, and user controllability.

Unlike the aforementioned approaches, EasyLAN goes beyond just hardcoding the examples in the prompts. EasyLAN explicitly adjusts the network architecture and the knowledge within each agent based on the provided examples. EasyLAN enables a more comprehensive utilization of the example data, maximizing the information extracted from the examples. It also gives developers a more intuitive understanding of what their systems have learned from the examples and simplifies debugging.

### 2.3 Human-Computer Collaborative Tools

In traditional human-computer collaboration models, the computer typically offers optional hints or suggestions, aiding the human user in task completion. In such systems, the human user shoulders the workload predominantly. These types of tools have been widely utilized across various domains, including but not limited to, patient note-taking [64, 70, 83], storytelling [59, 79], and other creative endeavors [13, 22, 30, 31, 65].

With the advancement of AI, an increasing number of tools replace human involvement entirely. An illustrative example is AI-assisted painting, which can autonomously generate a complete image from a user-provided text description or partially complete image [18, 21, 23, 27, 37, 48]. This autonomy, however, often comes at the cost of user agency in the creative process. Users frequently find themselves in a cycle of iteratively adjusting inputs [39, 66] or expending additional effort in interactive modifications to obtain desired outputs [17, 49]. This highlights a critical gap in balancing automation and user control in current AI systems.

Human-in-the-loop (HITL) [57, 85] can be considered a special kind of human-computer collaboration. Instead of the computers assisting humans, HITL emphasizes humans helping computing systems improve their capabilities with additional effort (e.g., providing annotations). However, in existing HITL systems, users cannot engage in the internal updating processes of the system. This results in a lack of interactivity [45] in HITL and makes it challenging to support high-level knowledge [73].

In this paper, we adopt a human-computer collaboration tool that diverges from the approaches above: the computer autonomously executes tasks (e.g., updating the LAN) while the human supervises its execution and intervenes if necessary. This approach not only alleviates the user's workload but also enhances the transparency and controllability of the computing system [72].

## 3 LLM AGENT NETWORK (LAN) DESIGN

A LAN is composed of multiple interconnected agents. The agents are responsible for sub-tasks and collaboratively accomplish a specific complex task. As illustrated in Figure 2(a), an agent comprises four components: an input module, a control module, an execution module, and an output module. The input module receives data from the output modules of predecessor agents, thereby connecting the agents into a directed acyclic graph (DAG). The control and execution modules constitute the core of an agent. They leverage LLMs to accomplish their functionalities and are subject to optimization during LAN updates.

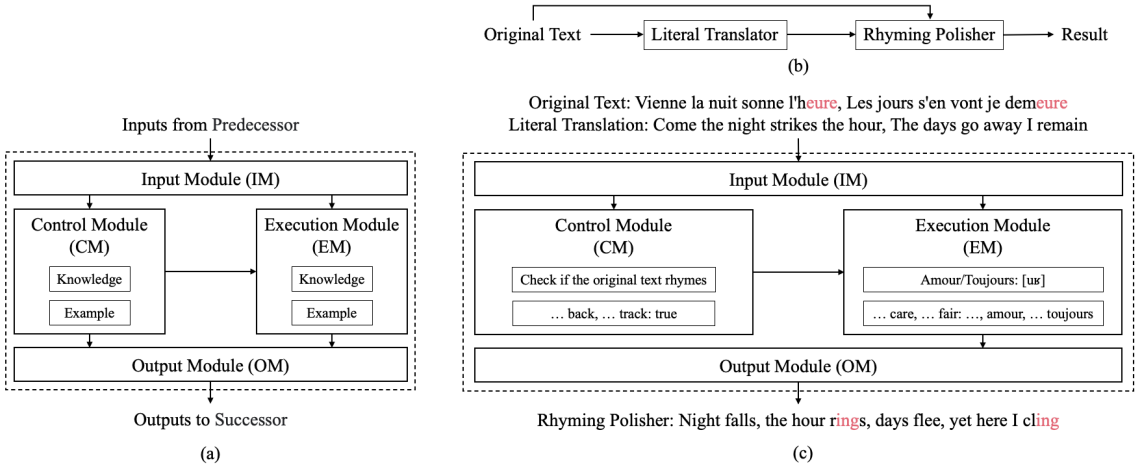



Fig. 2. An overview of an agent in a LAN. (a) The modules inside an agent. (b) an example LAN. "Literal Translator" and "Rhyming Polisher" are agents in the LAN. (c) Details of the "Rhyming Polisher" in (b). It receives the original French text and the output from the "Literal Translator" and computes a rhyming translation result."

### 3.1 Input Module (IM) & Output Module (OM)

The input module (IM) receives results from the predecessor agents' output modules (OMs). It also accepts external inputs (e.g., the French text awaiting translation in Figure 2(c)). The OM produces varying outputs based on the agent's activation status[1]:

(1) When the agent is not activated (i.e., the control module returns 'False'), the OM forwards the received inputs as its outputs.
(2) When the agent is activated (i.e., the control module returns 'True'), the OM outputs the results generated by the execution module.

### 3.2 Control Module (CM)

The Control Module (CM) serves as a decision-maker, akin to a "router", to evaluate whether the current agent should be activated. It facilitates conditional activation of agents, allowing the LAN to adapt its flow in real time by selecting a directed trail within the network. For instance, as illustrated in Figure 2(c), the 'Rhyming Polisher' is activated when the CM identifies rhyming elements in the French text input ("l'heure" and "demeure"). Conversely, for non-rhyming inputs, the CM would not activate the agent. By embedding such decision-making logic within the agent itself, instead of adding an additional agent to the network, we substantially simplify the LAN's architecture. The CM's decision impacts the final output of an agent, as discussed in Section 3.1.

Table 1 outlines the properties of a CM. When enabled and provided that all required predecessor agents are activated, the CM leverages LLMs to evaluate whether the agent should be activated. This assessment is based on the external input, outputs from upstream agents, and the specific subtask the agent is designed to undertake. For details of the prompt used in the CM, please refer to Section 5.1.

[1]The activation status will be further elaborated in Section 3.2.

Table 1. CM Properties and their explanations

| Property | Explanation |
|---|---|
| Enabled | Boolean. If this property is set to false, it indicates that the CM will activate the agent under any input.<br>This attribute is employed for critical agents within the LAN (e.g., Literal Translator) to reduce computational cost. |
| Required Predecessors | A list of predecessor agents. If any of these agents are not activated, the CM will not activate its agent. |
| Knowledge | A knowledge base. The CM utilizes the knowledge to determine whether to activate its agent.<br>For example, "If the original text exhibits rhyming, the current agent should be activated." (Rhyming Polisher in Figure 2(c)) |
| Examples | An example list. The CM utilizes the examples to determine whether to activate its agent.<br>An example is composed of the input of the agent and the CM's result. For example:<br>Input: (from LAN) Bras levés raides pour blâmer, Dans faux gestes d'aimer;<br>(from *Literal Translator*) Arms raised stiff to blame, In false gestures of loving.<br>Result: true |

### 3.3 Execution Module (EM)

The agent completes its subtask using the execution module (EM). Table 2 presents the properties of an EM. The EM utilizes LLMs to compute the agent's output based on the external input, outputs from upstream agents, and its designated task. For details of the prompt used in the CM, please refer to Section 5.1.

Table 2. EM Properties and their explanations. All the examples come from the EM of the "Literal Translator" in Figure 2

| Property | Explanation |
|---|---|
| Subtask Description | A string, describing the subtask the agent undertakes. For example: "Translating its literal meaning from the input into English" |
| Output Description | A string, describing the output of the agent. For example: "A string, indicating the literal translation" |
| Knowledge | A knowledge base. The EM utilizes the knowledge to execute its subtask. For example, "levés means lifted in English" |
| Examples | An example list. The EM utilizes the examples to decide how to execute its subtask.<br>An example is composed of the external output and the EM's result. For example:<br>Input: (from LAN) Bras levés raides pour blâmer, Dans faux gestes d'aimer;<br>(from *Literal Translator*) Arms raised stiff to blame, In false gestures of loving.<br>Result: Arms raised stiff to blame, In false gestures of loving |

## 4 SYSTEM DESIGN

An LLM agent network (LAN) decomposes a complex task into multiple sub-tasks, each of which is managed by an individual agent. EasyLAN facilitates the user in developing a LAN for a complex task. The user is required to provide only 1) a natural language description of the desired task and 2) a small set of training examples comprising both inputs and expected outputs. EasyLAN employs an iterative approach to update the LAN. Throughout this process, users supervise the automated behaviors executed by EasyLAN and intervene as necessary. This section is organized as follows: first, we elucidate the automated pipeline responsible for LAN updates (Section 4.1), followed by an introduction on how the user can exert influence over EasyLAN's workflow (Section 4.2).

### 4.1 Automated Update of LAN

*4.1.1 Initialization of the LAN.* The initial agent network is comprised solely of a single agent. Its functionality is delineated by the user-provided task description, and its output is directly the outcome of that specified task. However, the performance of a single agent is inherently circumscribed due to its limitation of generating only end-to-end output.

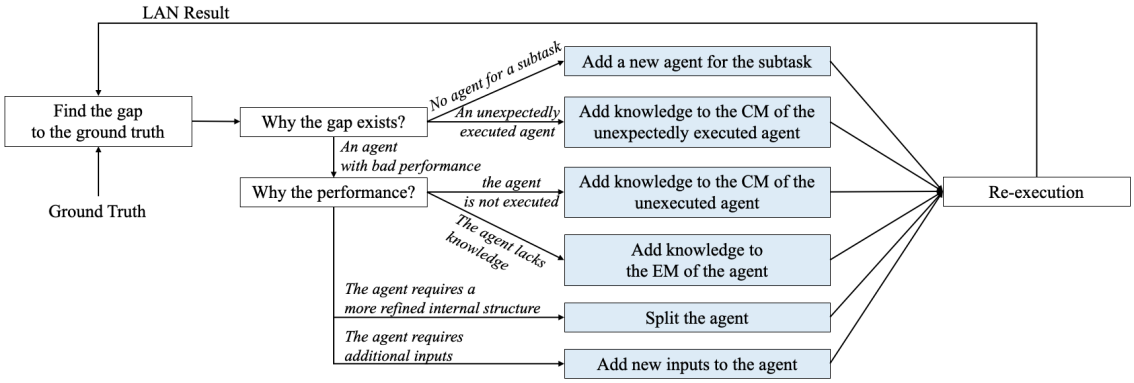



Fig. 3. The pipeline for updating the LAN. The light blue rectangles indicate update strategies.

*4.1.2 Pipeline to update a LAN.* Figure 3 outlines the pipeline for updating the LAN. The pipeline comprises multiple iterations until the LAN can proficiently handle the current training input. At this time, EasyLAN adds the inputs and outputs of each agent's CM and EM to the corresponding module's example library, further ensuring the performance of the LAN.

During each iteration, EasyLAN selects an update strategy from Table 3 following the four steps below:

(1) (Step 1) EasyLAN calculates the gap between the LAN output and the expected ground truth. If the LAN fails to yield satisfactory results, it indicates that at least one constituent sub-task has been inadequately executed. EasyLAN identifies the most significant deficiency in the LAN and isolates the crucial sub-task.
(2) (Step 2) EasyLAN analyzes the reasons for the gap's existence, which fall into three categories:
   - No agent is responsible for the sub-task.EasyLAN selects the strategy that creates a new agent and proceeds to Step 4.
   - The sub-task should not be executed, but the corresponding agent is activated. EasyLAN chooses the strategy that updates the agent's CM knowledge to deactivate it and then moves to Step 4.
   - An agent already manages the sub-task, but its performance is poor. EasyLAN proceeds to Step 3 to identify a more specific reason.
(3) (Step 3) EasyLAN investigates the reasons behind the agent's poor performance and advances to Step 4. These reasons can be classified into four possibilities:
   - The agent is not activated. EasyLAN selects the strategy to update the agent's CM knowledge to activate it.
   - The agent lacks knowledge. EasyLAN chooses the strategy to update the agent's EM knowledge to enhance its performance.
   - The agent requires a more refined internal structure. EasyLAN selects the strategy to split the agent into two or more agents.
   - The agent needs additional inputs from other agents. EasyLAN opts for the strategy that adds new edges to the network, allowing the agent to receive the necessary inputs.
(4) (Step 4) EasyLAN calculates the parameters for the chosen strategy. The LAN undergoes an update with this strategy, is re-executed to obtain a new result, and initiates a new iteration.

Table 3. Update strategies and the underlying causes of error to rectify

| Update Strategy | Cause of Error |
|---|---|
| Add an agent | No agent is responsible for a sub-task. |
| Split an agent | A task is already managed by an agent, but requires multiple steps or varying conditions to achieve a satisfactory output. |
| Add Knowledge to the CM | A task is already managed by an agent, but the agent is not activated. |
| | A sub-task that should not be executed is carried out, indicating that an agent is erroneously activated. |
| Add Knowledge to the EM | A task is already managed by an agent, but requires additional knowledge to yield satisfactory output. |
| Add inputs to an Agent | A task is already managed by an agent, but requires additional inputs from other agents to yield satisfactory output. |

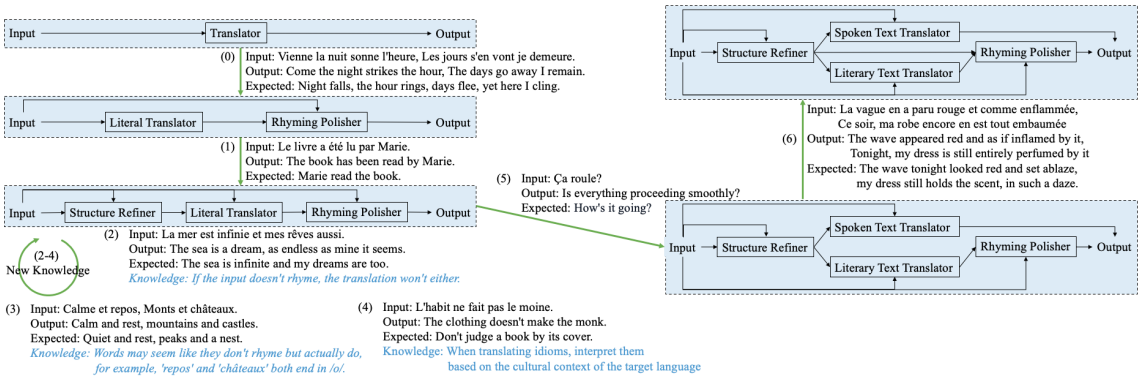



Fig. 4. An example of how EasyLAN updates a LAN. (0) EasyLAN decomposes the *Translator* into *Literal Translator* and *Rhyming Polisher* to ensure that the translation output can rhyme when necessary. (1) EasyLAN adds a *Structure Refiner* to adjust the syntactic structure of sentences (e.g., converting passive voice to active voice). (2) EasyLAN adds knowledge to the CM of the *Rhyming Polisher* to prevent unnecessary rhyming. (3) EasyLAN adds knowledge to the CM of the *Rhyming Polisher* to better identify whether the input sentence is rhyming. (4) EasyLAN adds knowledge to the EM of the *Literal Translator* to improve its capability to translate idioms. (5) EasyLAN splits the *Literal Translator* into *Spoken Text Translator* and *Literary Text Translator* to better cater to diverse translation needs. (6) EasyLAN adds a connection from the *Structure Refiner* to the *Rhyming Polisher* to ensure that the output from the *Rhyming Polisher* also adheres to the sentence structure defined by the *Structure Refiner*.

*4.1.3 Details of the update strategies.* This section focuses on the specifics of various error causes and their respective update strategies.

**No agent is responsible for a sub-task**. EasyLAN creates a new agent for the sub-task and positions the agent appropriately within the network. As depicted in Figure 4(1), French employs passive voice, which may result in redundancy when translated word-for-word into English. Therefore, EasyLAN adds a *Structure Refiner* into the LAN to explicitly assess the sentence structure of translated results (e.g., active or passive voice).

**A sub-task (handled by an agent) that should not be executed is carried out**. EasyLAN updates the knowledge of the CM to deactivate the mistakenly executed agent. As depicted in Figure 4(2), the input sentence lacks rhyme, yet the *Rhyming Polisher* is triggered and converts the output from the *Literal Translator* into rhyming text. Rhyming carries side effects that may lead to semantic changes. Therefore, EasyLAN augments the knowledge of the *Rhyming Polisher*'s CM to explicitly require that the *Rhyming Polisher* should not execute if the original text does not rhyme.

**A sub-task is already managed by an agent, but its performance is suboptimal**. EasyLAN assesses the reasons behind the bad performance and selects corresponding strategies:

(1) If the agent is not activated, EasyLAN updates the knowledge of its CM to activate it. As depicted in Figure 4(3), the LLM erroneously perceives "repos" and "châteaux" as ending with different vowels and deactivates the *Rhyming Polisher*. EasyLAN thus adds knowledge to the CM to enforce its activation.

(2) If the agent requires additional knowledge to yield satisfactory results, EasyLAN summarizes and adds the required knowledge to the EM of the agent. As illustrated in Figure 4(4), the input text carries specific cultural meanings ("one must not trust appearances"), yet the *Literal Translator*'s output lacks this nuance. Consequently, EasyLAN enriches the knowledge within the *Literal Translator*'s EM to enhance its performance.

(3) If the agent requires a more intricate structure, EasyLAN splits the agent into two or more separate agents and determines their interconnections. EasyLAN ensures that this division maintains the agent's semantic consistency. The knowledge of the original agent will be redistributed among the new agents. EasyLAN supports two kinds of division: (1) sequential division, which breaks down the sub-task into finer-grained steps (e.g., dividing the *Translator* into the *Literal Translate* and the *Rhyme Optimization*, as shown in Figure 4(0)); (2) parallel division, separates the agent based on distinct conditions. For example, in Figure 4(5), the *Literal Translator* is split into the *Literary Text Translator* and the *Spoken Text Translator* to handle different literary styles.

(4) If the agent requires additional inputs from other agents to yield satisfactory results, EasyLAN adds new edges to the network to ensure that the agent receives the necessary inputs. These additional inputs should already be computed within the agent network; otherwise, EasyLAN would create an agent responsible for their computation. As illustrated in Figure 4(6), although the *Structure Refiner* has already stipulated the use of an active voice in the translation result, this information was not conveyed to the *Rhyming Polisher*, leading to it once again altering the result to a passive voice. To rectify this issue, EasyLAN establishes a connection from the *Structure Refiner* to the *Rhyming Polisher*.

*4.1.4 Ensuring the accuracy of previous training inputs.* EasyLAN's application of an update strategy should not compromise the accuracy of past training inputs. We adopt the following approaches to ensure this:

(1) EasyLAN does not encompass disruptive update strategies, such as agent deletion, connection removal, knowledge removal, or task modification. The current LAN is generally satisfactory and can correctly process previous training inputs. These strategies remove useful components from the network without providing any compensation.

(2) When the update strategy is adding knowledge (whether to the CM or the EM), EasyLAN employs few-shot learning to ensure the accuracy of historical training samples. As mentioned in Section 4.1.2, when the LAN can correctly handle a training sample, its operational state (e.g., the inputs and outputs of each agent's CM and EM) is recorded and added to the example repository. Therefore, EasyLAN does not need to take any additional actions.

(3) When the update strategy is adding inputs, the few-shot learning mechanism still works. This is because the old inputs remain present, allowing the LAN to fully utilize the examples based on them. Therefore, EasyLAN does not need to take any additional actions.

(4) When the update strategy is adding an agent, EasyLAN adds negative examples to the new agent's CM to ensure that the new agent will not be activated when processing historical training inputs. By taking this additional

approach, we can ensure that the set of agents activated when handling historical inputs remains consistent, and therefore the final output also remains unchanged.

(5) When the update strategy is splitting an agent, EasyLAN's additional actions are quite complex. For a given historical input, if the agent being split should not be activated, we simply add negative examples to the CMs of the newly created agents to ensure they will not be activated either. When the agent being split needs to be activated, EasyLAN also splits the examples of the original agent:
   (a) Splitting into multiple parallel agents. EasyLAN utilizes LLM to select an agent from the new agents. Then EasyLAN adds examples to the CMs of the new agents so that only the selected agent will be activated when processing the given history input. Finally, EasyLAN adds a new example to the EM of the selected agent to ensure that the output of that agent matches the output of the original agent.
   (b) Splitting into multiple sequential agents. We only concern the output of the last agent. Therefore, EasyLAN executes this historical input in the new LAN: for non-last agents, EasyLAN adds the execution results as an example to the EMs; for the last agent, EasyLAN adds the execution result of the original agent to its EM.

Note that, as outlined in Section 4.2, developers have the capability to manually edit the LAN. Given the potential complexity of these manual interventions, automatically ensuring the accuracy of previous training examples falls outside the scope of this paper.

### 4.2 User Interaction with EasyLAN

User interaction is divided into two categories: (1) Supervision of the automatic update pipeline by EasyLAN: If users believe that EasyLAN has made errors during this process, they can manually intervene. EasyLAN will continue executing unfinished steps following user intervention. (2) Manual editing of the LAN: Users can edit the LAN at any time.

Figure 5 displays the interaction interface, featuring Region 1 with a diagram providing an overview of the structure of the LAN. Users can select any agent in the diagram, with its detailed information appearing in Region 2. Region 3 allows users to provide training samples (both input and output) to the system, while Region 4 enables users to monitor and intervene in EasyLAN's LAN updating process.

*4.2.1 Supervision and intervention in EasyLAN.* EasyLAN displays results at each step of the LAN update process. As shown in Figure 5-4, the most significant discrepancy between the LAN's output and the ground truth is whether the translation result maintains rhyme. EasyLAN first calculates the underlying cause of this issue, which is the absence of an agent called *PoeticTranslator*, and presents the result for user review. If the user believes the current step is correct, they can click "confirm" to proceed to the next step. In cases where the user finds EasyLAN's results unsatisfactory, they can intervene in two ways (separately or simultaneously, as shown in Figure 6) and click "retry" to have the system recalculate based on their intervention.

**Way 1: manual modification of EasyLAN's output**.As shown in Figure 5-4 & 6, we utilize a JSON editor to present the result of the current step, allowing the user to edit them directly within the editor. The user can focus solely on important details and use placeholders (<???>) in non-essential properties, which EasyLAN will auto-complete. We provide keyboard shortcuts to facilitate this process, enabling users to (1) click a field (e.g., agent_name in Figure 6) to set its value as a placeholder and (2) insert a placeholder where the cursor is. Additionally, we have implemented buttons

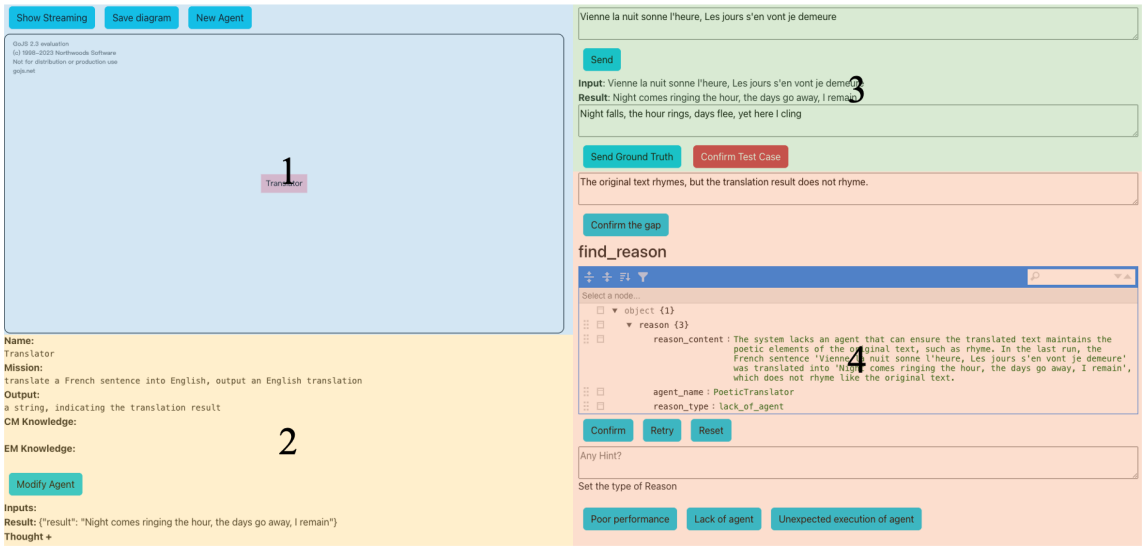



Fig. 5. The user interface. Region 1 allows users to inspect and modify the LAN structure. Agents can be selected by clicking the pink rectangles. Region 2 facilitates inspection and editing of the selected agent's properties. In Region 3, users can provide training examples to EasyLAN. Region 4 offers insights into and intervention options for EasyLAN's automated LAN update workflow.

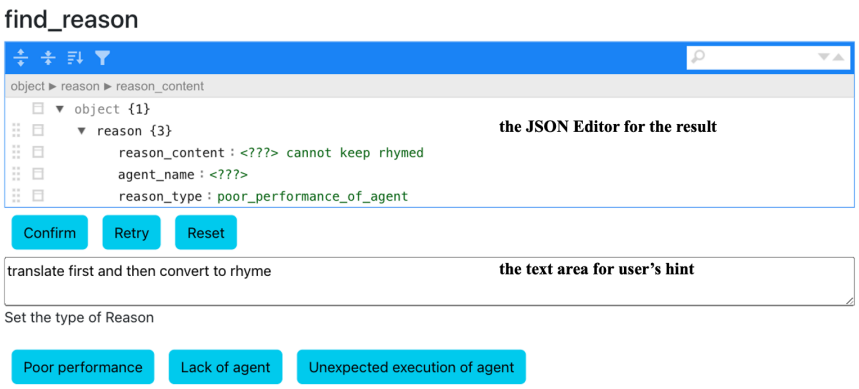



Fig. 6. How the user intervenes in the pipeline of EasyLAN updating the LAN. The user can directly modify the result within the JSON editor and utilize placeholders (<???>) to allow EasyLAN to auto-complete them. They can also enter hints in the text area to guide EasyLAN in adjusting the results.

to assist users in configuring the values of some key fields. For example, users can directly click "Poor performance" or "Lack of agents" to quickly set the value of the field "reason_type," which indicates the cause of the error.

**Way 2: providing hints for EasyLAN to adjust its computation**. In this approach, the user does not tediously modify the details of EasyLAN's results but can instead influence EasyLAN's computation results through concise, high-level natural language descriptions. This method offers significant advantages, especially when EasyLAN's computations are complex. For instance, when EasyLAN splits an agent, it calculates all the properties of the new agents (i.e., Table 1 & 2). If the user is dissatisfied with the result, directly modifying these properties, even a few, can be cumbersome.

In contrast, the user can directly describe their expected splitting result here (e.g., "translate first and then convert to rhyme") and request EasyLAN to retry accordingly.

*4.2.2 Manual modification of the LAN.* The user can manually edit the agent network in Region 1. We support four operations: (1) Creating a new agent by clicking the "New agent" button. Detailed information about the new agent can be modified in Region 2. (2) Deleting an agent by selecting it and pressing the "Delete" key on the keyboard. This action also removes both the incoming and outgoing edges of the agent. (3) Connecting two agents by dragging the starting agent onto the target agent. (4) Deleting a connection between agents by selecting the edge and pressing the "Delete" key on the keyboard. When the user selects an agent in the diagram in Region 1, they can manually edit the agent's name, task description, output description, and knowledge of CM and EM in Region 2.

EasyLAN automatically verifies the editing results. The LAN cannot be saved if any of the following conditions are satisfied: (1) the network contains cycles, (2) any agent's name, task description, or output description is empty, or (3) two agents have duplicate names.

## 5 IMPLEMENTATION

### 5.1 LAN execution

EasyLAN executes each agent sequentially following a topological order. The output of the last agent executed will serve as the LAN's final output. For each agent, EasyLAN utilizes LLMs to determine its activation status. If activated, it proceeds with result computation. Figure 7 illustrates the prompt's structure, which comprises: (1) a task description, specifying the LLM's current task; (2) agent inputs, encompassing the LAN's input, the sub-tasks of the agent's predecessors, and their corresponding outputs; (3) knowledge and examples (if available); (4) a zero-shot Chain of Thought prompt to extract the LLM's reasoning process; (5) a JSON template that defines the desired output format. It is important to note that the LLM may not consistently adhere to the format, in which case EasyLAN will call the LLM again to adjust the output to meet the specified format.

### 5.2 LAN Update

In this section, we focus on the implementation of each step in the LAN update process described in Section 4.1, especially the prompts EasyLAN uses.

**Description of the LAN**. We incorporate LAN descriptions into the prompts to provide the LLM with a comprehensive understanding of the LAN's content. This facilitates the LLM in better identifying its shortcomings. As depicted in Section A.1 in the Appendix, these descriptions encompass the following elements: (1) Information regarding all agents, including their names and task descriptions; (2) Data flow between agents, specifying the data's content, its source, and its destination. This data flow originates from the LAN's last execution and should be optimized to yield satisfactory LAN outputs. (3) The LAN's input and output.

**Description of the agent**. We include agent descriptions in the prompts to provide the LLM with comprehensive insights into the agent's specifics. This aids the LLM in making better determinations regarding agent modifications. As depicted in Section A.2 in the Appendix, agent descriptions encompass the following elements: (1) agent information, including the agent's name, task description, and **output description**; (2) **knowledge and examples within the agent's CM**; (3) **knowledge and examples within the agent's EM**; (4) agent's inputs, **thought processes (involving both CM and EM)**, and outputs from the previous execution. The content in bold was not included when describing the LAN.

```
// task description
You are the control module of an agent in an LLM agent network.
The goal of the network is <task description>.
The subtask of the agent: <subtask description of the agent>.
You need to determine if the following agent should execute.
// inputs
The input of the network: <LAN input>.
Here are the inputs for you:
1. <agent name>: <agent task>. Its result: <agent result>.
2. ...
// knowledge and examples
You have the following pieces of knowledge:
1. <CM knowledge 1>.
2. ...
You have the following examples:
1. <CM example 1>.
2. ...
// chain of thought prompt
You should first think step by step and output your reasoning process.
// outputs and JSON template
Here are the requirements for your output:
"true" means the agent should execute, and "false" means the opposite.
You should finally generate your result according to the format:
{"result": <true/false>}
```

```
// task description
You are the execution module of an agent in an LLM agent network.
The goal of the network is <task description>.
The subtask of the agent: <subtask description of the agent>.
You need to carry out the subtask mentioned above.
// inputs
The input of the network: <LAN input>.
Here are the inputs for you:
1. <agent name>: <agent task>. Its result: <agent result>.
2. ...
// knowledge and examples
You have the following pieces of knowledge:
1. <EM knowledge 1>.
2. ...
You have the following examples:
1. <EM example 1>.
2. ...
// chain of thought prompt
You should first think step by step and output your reasoning process.
// outputs and JSON template
Here are the requirements for your output:
<output description>
You should finally generate your result according to the format:
{"result": ...}
```

Fig. 7. The prompts used in agent execution. Left: the prompt used by the CM to determine whether the agent should be activated. Right: the prompt used by the EM to calculate the output of the agent. Lines starting with a double slash ("//") are comments. Angle brackets (<>) in the prompts denote placeholders that should be replaced with actual values.

**Update steps**. As shown in Table 4, the prompts used in each update step consist of the following components: (1) the task to be accomplished in the current step, which aligns with the discussion in Section 4.1; (2) the outputs of previous steps; (3) a description of the LAN or a specific agent to be updated; (4) a JSON template to specify the LLM's output format. We provide a detailed explanation of this template within the prompt; please refer to Section A in the Appendix for more details.

Table 4. The prompts used in the update steps. Please refer to Section A in the Appendix for more details. We omit some punctuation marks in the JSON template.

| | Task for the Step | Inputs from Previous Steps | LAN Description | Agent Description | JSON Template |
|---|---|---|---|---|---|
| Step 1 | "Find the gap between the LAN's output and the ground truth" | - | Yes | No | "gap": ... |
| Step 2 | "Find why the gap exists" | 1. the gap | Yes | No | "reason_type": ... "agent_name: ... "reason_content": ... |
| Step 3 | "Why the agent has a poor performance" | 1. description of the poor performance | No | Yes | "reason_type": ... "reason_content": ... |
| Step 4 | "Calculate the parameter for the strategy" | 1. the agent to be updated (if any) 2. the selected strategy | Yes | Yes | "parameters": ... |

### 5.3 System Implementation

We implemented the server back end using Flask and made remote calls to OpenAI's GPT-4-0613. The connection to OpenAI can sometimes be unstable, although this issue is somewhat mitigated after we enabled the "stream" attribute [2]. If the connection breaks down, we continue to retry until OpenAI returns a complete result. We employed React to implement the front end in the browser, and GoJS was utilized for rendering the diagram in Region 1.

## 6 EVALUATION STUDY

### 6.1 Experimental Tasks

Table 5 presents the four experimental tasks for the evaluation study. The first was used for the tutorial, and the remaining three were for the formal study. Given that subjective opinions on these tasks could vary among participants, we aimed to reduce variability by establishing well-defined criteria for each task. These criteria were used to evaluate the performance of the LAN. The construction of training samples is beyond the scope of this paper. In this experiment, we constructed 16 training examples for each task, all conformed to the abovementioned criteria. These examples were then split randomly into two subsets: one for training (8 examples) and another for testing (8 examples).

While these tasks are all typical natural language processing tasks, they vary in difficulty for LLMs. LLMs [8, 55] are trained on multilingual data and excel at translation. However, researchers do not specifically collect sentence compression and couplet generation data to train LLMs. Besides, these tasks are not downstream training objectives for LLMs, making them more challenging.

Table 5. Tasks in the user study. Please refer to Section B in the Appendix for more details about the tasks.

| Task ID | Task Description | Criteria |
|---|---|---|
| Tutorial | Grammatical error detection [40] | 1. Identify subject-verb agreement errors<br>2. Identify missing sentence components<br>3. Identify redundancy in the sentence<br>4. Identify ambiguity in the sentence |
| 1 | Chinese-English translation [67] | 1. The result is a grammatically correct sentence.<br>2. Correctly chooses whether to rhyme or not.<br>3. Accurately translates words with multiple meanings.<br>4. Correctly interprets metaphors specific to the Chinese context. |
| 2 | Sentence compression [84] | 1. The result is a grammatically correct sentence.<br>2. Remove all the attributives from the sentence.<br>3. Remove all the adverbials from the sentence.<br>4. Remove all the complements from the sentence. |
| 3 | Chinese couplet generation [7] | 1. Same character count for both couplets.<br>2. Matching emotion and related themes.<br>3. Same parts of speech for corresponding words.<br>4. No repeating characters between couplets. |

### 6.2 Participants

We recruited 12 participants (6 males and 6 females, ages 18-29). All were familiar with the aforementioned tasks, and none were programmers.

[2] https://platform.openai.com/docs/api-reference/completions/create#stream

### 6.3 Baseline

We use EasyLAN with its automatic update pipeline disabled as our baseline. Participants can only manually modify the LAN in Regions 1 and 2 (Figure 5). They can provide inputs to the LAN and obtain outputs in Region 3. We disable Region 4 in the baseline to prohibit any user interactions.

### 6.4 Procedure

Participants were randomly assigned to two groups to control for individual differences: one group used EasyLAN, while the other group used the baseline system. We provided participants with a system briefing and explained the experiment's tasks. During the tutorial task, participants received guidance from experimenters and had the opportunity to ask questions.

In the formal study, participants completed three tasks in a randomized order. Training examples were presented randomly, and participants were instructed to modify the LAN until the system's outputs met predefined criteria. The entire experiment lasted approximately 1.5 hours. Afterward, we administered questionnaires to collect subjective feedback.

After conducting experiments with all participants, we fed the testing examples into the user-constructed LANs. during the offline evaluation. The results were then shuffled and independently assessed by two experimenters to determine if they met the predefined criteria. In cases where the opinions diverged, a conclusion was reached through further discussion.

### 6.5 Results

#### *6.5.1 Terminology and metrics.*

**A LAN modification** is defined as the aggregate of all updates executed between two successive runs of the LAN. This encapsulates the implementation of a comprehensive LAN update plan and may contain multiple operational actions (e.g., creating a new agent and then editing its attributes).

**User editing distance** (UED): A metric used to estimate the "quantity" of user interactions. We consider a single mouse click or pressing a key (e.g., typing a letter) to have an editing distance of 1. Dragging and text selection are assigned an editing distance of 2, as the user needs to determine both a starting and ending point. Therefore, deleting a text segment has an editing distance of 3, which involves selecting the text first and pressing the "Delete" key.

**LAN modification distance** (LMD): A metric designed to quantify the differences between two LANs. Let A and B represent two different LANs. If a user manually edits A into B solely through interactions supported by the baseline, the *theoretical minimum* UED required for this transformation is designated as the LMD between A and B. The LMD of a task is defined as the LMD between the initial LAN and the final LAN corresponding to that task.

**Interaction time**: Defined as the total time minus the LLM execution time. The running of LLM is computationally expensive, and optimizing its efficiency falls outside the scope of this paper. The exclusion of LLM's execution time allows for a more accurate measure of EasyLAN's operational efficiency.

**LAN output scores**: As discussed in Section B, we propose criteria for all tasks. The score of a LAN output is 1 if it fulfills all the criteria, else 0. The score of a LAN is denoted as the average score of its output.

**Naive LAN**: A LAN with only one agent. The agent relies solely on few-shot learning, meaning that its EM knowledge is exactly the training examples.

### 6.6 User behaviors

EasyLAN offers a reduced interaction burden compared to the baseline. Table 6 displays the frequency of different editing actions and their average UEDs. In EasyLAN, the UED to complete a task is 374 (47 per training example), and that for the baseline is 947 (118 per training example, $p < 0.001$). This suggests that EasyLAN enables users to develop a satisfactory LAN with 39.5% of interactions, which is mainly due to two key factors. (1) EasyLAN can amplify user actions. The average LMD per task in EasyLAN is 2424, 6.48 times greater than the corresponding UED. The LMDs for EasyLAN are very large, mainly because the LLM tended generate very long sentences as knowledge. (2) In the baseline system, the LMD is slightly less than the UED (825 < 945), which indicates that manual edits always introduce errors, necessitating additional corrective actions.

Table 6. User actions and their average UED. The UED of EasyLAN decreases by 60.5% relative to that of the baseline.

| Actions | Baseline | | | EasyLAN | | |
|---|---|---|---|---|---|---|
| | Count per task | UED per action | UED per task | Count per task | UED per action | UED per task |
| Modify Step1 | 0 | 0 | 0 (0%) | 3.33 | 82.4 | 301 (80.6%) |
| Modify Step2 | 0 | 0 | | 1.00 | 2.61 | |
| Modify Step3 | 0 | 0 | | 0.94 | 1 | |
| Modify Step4 | 0 | 0 | | 1.39 | 17.0 | |
| New Agent | 1.56 | 1.00 | 946 (100%) | 0 | 0 | 72.3 (19.7%) |
| Modify Agent | 11.6 | 80.8 | | 1.67 | 41.9 | |
| Modify Edge | 2.11 | 3.16 | | 0.33 | 7.00 | |

EasyLAN demonstrated a lower frequency of manual modifications, implying that it helped users determine the correct modification plan more quickly. EasyLAN users conducted 8.22 modifications per task, whereas baseline users conducted 10.83 modifications. The average number of modifications per sample for EasyLAN is significantly lower than that for the baseline ($p=0.048 < 0.05$), suggesting that EasyLAN aided users in making more accurate adjustments to the LAN.

EasyLAN exhibits higher interaction efficiency. Users completed a task with EasyLAN in an average interaction time of 605 seconds and saved 39.3% of the time compared with the baseline (997 seconds, $p < 0.001$). EasyLAN exhibits varying degrees of time savings in different tasks. In Task 2 (682 seconds vs. 1046 seconds, $p < 0.001$) and Task 3 (551 seconds vs. 1325 seconds, $p < 0.001$), it saved 34.8% and 58.4% of time, significantly enhancing interaction efficiency. However, in Task 1, although the average interaction time decreased slightly compared to the baseline (581 seconds vs 620 seconds, 6.3%), statistical tests did not show significance ($p = 0.76$). The main reason is that Task 1 is a translation task in which LLM excels, requiring only minor LAN modifications to achieve the desired results. Consequently, the baseline's interaction time is small. In contrast, LLM struggles with Task 2 and Task 3, leading users to make substantial LAN modifications, thus resulting in a significant difference in interaction time between EasyLAN and the baseline.

### 6.7 Performance of the LANs

LANs constructed using EasyLAN have a 39.8% improvement in scores on training samples compared to the baseline (0.583 vs. 0.417, $p=0.009 < 0.05$). Note that the score of the naive LAN is 0.29 and the tasks are quite difficult for the LLM. The differences mainly exist in Task 1 (0.686 vs. 0.396) and Task 3 (0.563 vs. 0.375). However, the performance of EasyLAN and the baseline only differs slightly in Task 2 (0.500 vs. 0.479). A possible explanation is that Task 3 is quite

suitable for few-shot learning, as the score of the naive LAN in Task 3 is 0.5. Baseline users tended to add the training examples into the LAN as knowledge (we will discuss this in the next paragraph), which explains its performance.

We conducted a thorough comparison of the LANs generated by the two systems and identified the following differences:

(1) LANs constructed with EasyLAN exhibit a higher number of agents compared to the baseline system (2.89 vs 2.33, $p=0.04 < 0.05$), suggesting that EasyLAN breaks down complex tasks into more detailed components. This is likely because creating a new agent in the baseline system is a cumbersome process for users, as it requires them to manually define various attributes of the agent and edit the values in the GUI.

(2) The knowledge in the LANs constructed by EasyLAN is more general than that of LANs constructed with the baseline. All of the 121 pieces of knowledge from LANs built with EasyLAN contain both a general description and a concrete example. In contrast, the LAN constructed by the baseline system contains 98 pieces of knowledge, of which 33 (33.7%) are directly the given training examples (or part of them), without any generalization. Users of the baseline system often found it challenging to generalize knowledge from the training examples, and they tended to add the examples to the knowledge base directly. Although the correctness of the LAN can be ensured during the training process, it is difficult to guarantee the generalization ability.

### 6.8 Subjective Feedback

Table 7. Subjective feedback

| | Baseline | EasyLAN | P-value |
|---|---|---|---|
| Determining the modification plan for the agent network is easy. | 3.67 | 5.67 | 0.02 |
| Applying the modification plan for the agent network is fast | 4.17 | 6.00 | 0.03 |
| Interaction with the system is straightforward. | 4.17 | 6.33 | 0.002 |
| The system is intelligent, and you receive many hints from the system. | 4.00 | 6.67 | 0.007 |
| Willing to use this system to construct multi-agent networks. | 4.17 | 6.17 | 0.01 |

Table 7 compares user feedback for two systems. Overall, users have a positive attitude towards EasyLAN, considering it more intelligent and providing a relaxed and fast interaction experience.

## 7 DISCUSSION

### 7.1 The Difference Between Intra-Agent Knowledge and Inter-Agent Structures

In tutorials of the evaluation study, a frequently asked question is, "When should I add knowledge, and when should I create a new agent?" In fact, the internal structure of a LAN can be understood as encapsulating a form of knowledge that specifies "what actions to take" (hereafter termed W-knowledge). On the other hand, another form of knowledge outlines "how to execute those actions" (hereafter termed H-knowledge). Differentiating between these two categories of knowledge is crucial, as they present distinct characteristics:

(1) W-knowledge exhibits complex interrelationships, as distinct steps required for accomplishing an intricate task are often interdependent. In the context of LLMs, which solely accept natural language inputs, incorporating multiple pieces of W-knowledge into a single agent necessitates additional natural language descriptions to articulate their interrelationships. Notably, these descriptions serve not as imperatives but as optional constraints.

By making the interdependencies among W-knowledge explicit in the structure of the LAN, we achieve a more transparent representation of the hierarchical organization of the subtasks.

(2) W-knowledge is primarily task-oriented, whereas H-knowledge is predominantly input-related. The vast majority of agents are activated during LAN execution, which means that most of the W-knowledge within the LAN structure is fully leveraged, whatever the input is. On the other hand, the relevance of H-knowledge is dependent on specific user input, as illustrated in Table 2; for example, the piece of knowledge "levés means lifted in English" is not necessary when the input does not contain "levés". Given the limitations in prompt length and computational capabilities of LLMs, it is impractical to indefinitely incorporate knowledge and expect multitasking proficiency in a single text completion session. To optimize H-knowledge, one can filter relevant pieces based on the input. For managing W-knowledge, a proven strategy, as demonstrated by EasyLAN, is distributing it among a specialized network of agents, essentially transforming a single agent into a multi-agent network.

However, representing W-knowledge through agents and network architecture imposes a trade-off in computational efficiency. Extending the prompt with an additional sentence generally exerts minimal impact on operational efficiency, whereas invoking an extra LLM computation escalates computational overhead. In our user study, we noted that developers adapt their strategies based on specific scenarios. Initially, they may incorporate W-knowledge into an existing agent; if this fails to enhance the LAN's performance, they adjust the network architecture by introducing new agents or subdividing the existing ones.

### 7.2 Capability Boundary of the LAN: the Control Granularity Challenge

Although users can break down complex tasks into smaller granules to improve the capabilities of LLMs, there are inherent limitations to such granularity. Users could not break down tasks into small granularities that are not practical or meaningful in regular human cognition. A typical example is the couplet generation task. Couplets require that the number of characters in the upper and lower halves match. However, one user found that the LLM's ability to count the number of characters was unstable. This problem might be attributed to the process of encoding and tokenization, which is beyond the user's knowledge. The user could not address this issue within the multi-agent framework and complained, "If it can't even get this simple thing right, I don't know how to fix it." This indicates that optimizing fundamental tasks like counting numbers exceeds the capability of EasyLAN because those tasks fall below the minimum granularity where humans and LLMs can understand each other.

## 8 LIMITATION & FUTURE WORK

The construction of appropriate training examples falls outside the scope of this paper. Future work could qualitatively and quantitatively investigate the relationship between the features of training examples and the performance of EasyLAN. Additionally, employing natural language processing techniques for the automated generation or augmentation of training data [11] represents a promising avenue for future research.

EasyLAN can only handle acyclic networks, which implies that it cannot accommodate tasks requiring recursion and backtracking. Cycles significantly increase network complexity, as the same agent may be activated multiple times. EasyLAN faces challenges when confronted with excessively intricate network structures due to the long-distance dependency problem of LLMs. Future work may draw inspiration from existing neural network update strategies (e.g., backpropagation) to handle larger-scale networks.

Currently, EasyLAN cannot learn from users to improve its performance[50]. Future work can explore how EasyLAN continuously accumulates knowledge and adapts its pipeline during user interactions. This approach would enable EasyLAN to autonomously generate a LAN without requiring any user intervention.

EasyLAN is a collaborative editing tool, not a LAN generation tool. Therefore, it relies on LAN developers rather than on carefully designed strategies to ensure that the constructed LAN is optimal. Future work can focus on improving the LAN's performance by (1) minimizing the number of agents while maintaining performance to improve computational efficiency and (2) adding runtime error monitoring and handling mechanisms to prevent error propagation.

## 9 CONCLUSION

In this paper, we propose EasyLAN to assist users in developing LLM agent networks for their complex tasks. The most significant feature of EasyLAN is its ability to transform a single agent into an agent network using a small number of training samples. Users monitor and intervene in this process to ensure the accuracy and functionality of EasyLAN. The evaluation study demonstrates that EasyLAN can help users reduce interaction time by 39.3% while improving the performance of the constructed LANs by 39.8%. EasyLAN's contributions are two-fold: (1) a novel approach to constructing LLM networks that eliminates the need for users to design network structures and provides assistance during network updates; (2) a novel human-computer collaboration paradigm, "computer execution, user supervision", that reduces the user's burden while ensuring the computational system's correctness. We hope that EasyLAN's preliminary exploration of the LLM agent network can inspire new applications and human-computer collaborative systems in the rapidly evolving era of AI.

## REFERENCES


[1] Elena Agapie, Lucas Colusso, Sean A Munson, and Gary Hsieh. 2016. Plansourcing: Generating behavior change plans with friends and crowds. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*. 119–133.

[2] Sweta Agrawal, Chunting Zhou, Mike Lewis, Luke Zettlemoyer, and Marjan Ghazvininejad. 2022. In-context examples selection for machine translation. *arXiv preprint arXiv:2212.02437* (2022).

[3] Samreen Anjum, Chi Lin, and Danna Gurari. 2021. CrowdMOT: Crowdsourcing strategies for tracking multiple objects in videos. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW3 (2021), 1–25.

[4] Michael S Bernstein, Greg Little, Robert C Miller, Björn Hartmann, Mark S Ackerman, David R Karger, David Crowell, and Katrina Panovich. 2010. Soylent: a word processor with a crowd inside. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*. 313–322.

[5] Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, et al. 2023. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on Robot Learning*. PMLR, 287–318.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[7] Kuan-Yu Chiang, Shihao Lin, Joe Chen, Qian Yin, and Qizhen Jin. 2021. TransCouplet: Transformer based Chinese Couplet Generation. *arXiv preprint arXiv:2112.01707* (2021).

[8] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[9] John Joon Young Chung, Wooseok Kim, Kang Min Yoo, Hwaran Lee, Eytan Adar, and Minsuk Chang. 2022. TaleBrush: Sketching stories with generative pretrained language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–19.

[10] Damai Dai, Li Dong, Yaru Hao, Zhifang Sui, Baobao Chang, and Furu Wei. 2021. Knowledge neurons in pretrained transformers. *arXiv preprint arXiv:2104.08696* (2021).

[11] Haixing Dai, Zhengliang Liu, Wenxiong Liao, Xiaoke Huang, Yihan Cao, Zihao Wu, Lin Zhao, Shaochen Xu, Wei Liu, Ninghao Liu, et al. 2023. AugGPT: Leveraging ChatGPT for Text Data Augmentation. *arXiv preprint arXiv:2302.13007* (2023).

[12] Hai Dang, Lukas Mecke, Florian Lehmann, Sven Goller, and Daniel Buschek. 2022. How to Prompt? Opportunities and Challenges of Zero- and Few-Shot Learning for Human-AI Interaction in Creative Applications of Generative Models. arXiv:2209.01390 [cs.HC]

[13] Nicholas Davis, Chih-PIn Hsiao, Kunwar Yashraj Singh, Lisa Li, and Brian Magerko. 2016. Empirically studying participatory sense-making in abstract drawing with a co-creative cognitive agent. In *Proceedings of the 21st International Conference on Intelligent User Interfaces*. 196–207.

[14] Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric P Xing, and Zhiting Hu. 2022. Rlprompt: Optimizing discrete text prompts with reinforcement learning. *arXiv preprint arXiv:2205.12548* (2022).

[15] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey for in-context learning. *arXiv preprint arXiv:2301.00234* (2022).

[16] Dheeru Dua, Shivanshu Gupta, Sameer Singh, and Matt Gardner. 2022. Successive Prompting for Decomposing Complex Questions. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 1251–1265. https://doi.org/10.18653/v1/2022.emnlp-main.81

[17] Ahmed Elgohary, Christopher Meek, Matthew Richardson, Adam Fourney, Gonzalo Ramos, and Ahmed Hassan Awadallah. 2021. NL-EDIT: Correcting semantic parse errors through natural language interaction. *arXiv preprint arXiv:2103.14540* (2021).

[18] Judith E Fan, Monica Dinculescu, and David Ha. 2019. Collabdraw: an environment for collaborative sketching with an artificial agent. In *Proceedings of the 2019 on Creativity and Cognition*. 556–561.

[19] Alexander J Fiannaca, Chinmay Kulkarni, Carrie J Cai, and Michael Terry. 2023. Programming without a Programming Language: Challenges and Opportunities for Designing Developer Tools for Prompt Programming. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–7.

[20] Shivam Garg, Dimitris Tsipras, Percy S Liang, and Gregory Valiant. 2022. What can transformers learn in-context? a case study of simple function classes. *Advances in Neural Information Processing Systems* 35 (2022), 30583–30598.

[21] Leon A Gatys, Alexander S Ecker, Matthias Bethge, Aaron Hertzmann, and Eli Shechtman. 2017. Controlling perceptual factors in neural style transfer. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3985–3993.

[22] Matthew Guzdial, Nicholas Liao, Jonathan Chen, Shao-Yu Chen, Shukan Shah, Vishwa Shah, Joshua Reno, Gillian Smith, and Mark O Riedl. 2019. Friend, collaborator, student, manager: How design of an ai-driven game level editor affects creators. In *Proceedings of the 2019 CHI conference on human factors in computing systems*. 1–13.

[23] David Ha and Douglas Eck. 2017. A neural representation of sketch drawings. *arXiv preprint arXiv:1704.03477* (2017).

[24] Ziyao He, Yunpeng Song, Shurui Zhou, and Zhongmin Cai. 2023. Interaction of Thoughts: Towards Mediating Task Assignment in Human-AI Cooperation with a Capability-Aware Shared Mental Model. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–18.

[25] Jie Huang and Kevin Chen-Chuan Chang. 2022. Towards reasoning in large language models: A survey. *arXiv preprint arXiv:2212.10403* (2022).

[26] Ellen Jiang, Kristen Olson, Edwin Toh, Alejandra Molina, Aaron Donsbach, Michael Terry, and Carrie J Cai. 2022. Promptmaker: Prompt-based prototyping with large language models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. 1–8.

[27] Pegah Karimi, Mary Lou Maher, Nicholas Davis, and Kazjon Grace. 2019. Deep learning in a computational model for conceptual shifts in a co-creative design system. *arXiv preprint arXiv:1906.10188* (2019).

[28] Harmanpreet Kaur, Alex C Williams, Anne Loomis Thompson, Walter S Lasecki, Shamsi T Iqbal, and Jaime Teevan. 2018. Creating better action plans for writing tasks via vocabulary-based planning. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 1–22.

[29] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2022. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406* (2022).

[30] Jeongyeon Kim, Yubin Choi, Minsuk Kahng, and Juho Kim. 2022. FitVid: Responsive and flexible video content adaptation. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–16.

[31] Joy Kim, Mira Dontcheva, Wilmot Li, Michael S Bernstein, and Daniela Steinsapir. 2015. Motif: Supporting novice creativity through expert patterns. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. 1211–1220.

[32] Joy Kim, Sarah Sterman, Allegra Argent Beal Cohen, and Michael S Bernstein. 2017. Mechanical novel: Crowdsourcing complex work through reflection and revision. In *Proceedings of the 2017 acm conference on computer supported cooperative work and social computing*. 233–245.

[33] Tae Soo Kim, DaEun Choi, Yoonseo Choi, and Juho Kim. 2022. Stylette: Styling the web with natural language. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–17.

[34] Sandeep Kochhar and Mark Friedell. 1990. User control in cooperative computer-aided design. In *Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology*. 143–151.

[35] Sandeep Kochhar, Mark Friedell, Joe Marks, Steve Sistare, and Louis Weitzman. 1994. Interaction paradigms for human-computer cooperation in design. In *Conference companion on Human factors in computing systems*. 187–188.

[36] Andrew K Lampinen, Ishita Dasgupta, Stephanie CY Chan, Kory Matthewson, Michael Henry Tessler, Antonia Creswell, James L McClelland, Jane X Wang, and Felix Hill. 2022. Can language models learn from explanations in context? *arXiv preprint arXiv:2204.02329* (2022).

[37] Yuyu Lin, Jiahao Guo, Yang Chen, Cheng Yao, and Fangtian Ying. 2020. It is your turn: Collaborative ideation with a co-creative robot through sketch. In *Proceedings of the 2020 CHI conference on human factors in computing systems*. 1–14.

[38] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2021. What Makes Good In-Context Examples for GPT-3? *arXiv preprint arXiv:2101.06804* (2021).

[39] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D Gordon. 2023. "What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–31.

[40] Zhenghao Liu, Xiaoyuan Yi, Maosong Sun, Liner Yang, and Tat-Seng Chua. 2021. Neural quality estimation with multiple hypotheses for grammatical error correction. *arXiv preprint arXiv:2105.04443* (2021).

[41] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2021. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. *arXiv preprint arXiv:2104.08786* (2021).

[42] Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. 2022. Locating and editing factual associations in GPT. *Advances in Neural Information Processing Systems* 35 (2022), 17359–17372.

[43] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837* (2022).

[44] Meredith Ringel Morris, Jaime Teevan, and Katrina Panovich. 2010. What do people ask their social networks, and why? A survey study of status message Q&A behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 1739–1748.

[45] Eduardo Mosqueira-Rey, Elena Hernández-Pereira, David Alonso-Ríos, José Bobes-Bascarán, and Ángel Fernández-Leal. 2023. Human-in-the-loop machine learning: A state of the art. *Artificial Intelligence Review* 56, 4 (2023), 3005–3054.

[46] Prasanth Murali, Ian Steenstra, Hye Sun Yun, Ameneh Shamekhi, and Timothy Bickmore. 2023. Improving multiparty interactions with a robot using large language models. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–8.

[47] Joe O'Connor and Jacob Andreas. 2021. What Context Features Can Transformer Language Models Use? *arXiv preprint arXiv:2106.08367* (2021).

[48] Changhoon Oh, Jungwoo Song, Jinhan Choi, Seonghyeon Kim, Sungwoo Lee, and Bongwon Suh. 2018. I lead, you help but only with enough details: Understanding user experience of co-creation with artificial intelligence. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–13.

[49] Lihang Pan, Chun Yu, Zhe He, and Yuanchun Shi. 2023. A Human-Computer Collaborative Editing Tool for Conceptual Diagrams. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–29.

[50] Lihang Pan, Chun Yu, JiaHui Li, Tian Huang, Xiaojun Bi, and Yuanchun Shi. 2022. Automatically generating and improving voice command interface from operation sequences on smartphones. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–21.

[51] Shuofei Qiao, Yixin Ou, Ningyu Zhang, Xiang Chen, Yunzhi Yao, Shumin Deng, Chuanqi Tan, Fei Huang, and Huajun Chen. 2022. Reasoning with language model prompting: A survey. *arXiv preprint arXiv:2212.09597* (2022).

[52] Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–7.

[53] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2021. Learning to retrieve prompts for in-context learning. *arXiv preprint arXiv:2112.08633* (2021).

[54] Jeffrey M Rzeszotarski and Meredith Ringel Morris. 2014. Estimating the social costs of friendsourcing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2735–2744.

[55] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100* (2022).

[56] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*. PMLR, 31210–31227.

[57] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. 2020. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems* 33 (2020), 3008–3021.

[58] Hongjin Su, Jungo Kasai, Chen Henry Wu, Weijia Shi, Tianlu Wang, Jiayi Xin, Rui Zhang, Mari Ostendorf, Luke Zettlemoyer, Noah A Smith, et al. 2022. Selective annotation makes language models better few-shot learners. *arXiv preprint arXiv:2209.01975* (2022).

[59] Anne Sullivan, Mirjam Palosaari Eladhari, and Michael Cook. 2018. Tarot-based narrative generation. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*. 1–7.

[60] Jaime Teevan, Shamsi T Iqbal, Carrie J Cai, Jeffrey P Bigham, Michael S Bernstein, and Elizabeth M Gerber. 2016. Productivity decomposed: Getting big things done with little microtasks. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. 3500–3507.

[61] Jaime Teevan, Daniel J Liebling, and Walter S Lasecki. 2014. Selfsourcing personal tasks. In *CHI'14 Extended Abstracts on Human Factors in Computing Systems*. 2527–2532.

[62] Miles Turpin, Julian Michael, Ethan Perez, and Samuel R Bowman. 2023. Language Models Don't Always Say What They Think: Unfaithful Explanations in Chain-of-Thought Prompting. *arXiv preprint arXiv:2305.04388* (2023).

[63] Bryan Wang, Gang Li, and Yang Li. 2023. Enabling conversational interaction with mobile ui using large language models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–17.

[64] Dakuo Wang, Liuping Wang, Zhan Zhang, Ding Wang, Haiyi Zhu, Yvonne Gao, Xiangmin Fan, and Feng Tian. 2021. "Brilliant AI doctor" in rural clinics: Challenges in AI-powered clinical decision support system deployment. In *Proceedings of the 2021 CHI conference on human factors in computing systems*. 1–18.

[65] Hao-Chuan Wang, Dan Cosley, and Susan R Fussell. 2010. Idea expander: supporting group brainstorming with conversationally triggered visual thinking stimuli. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*. 103–106.

[66] Yunlong Wang, Shuyuan Shen, and Brian Y Lim. 2023. RePrompt: Automatic Prompt Editing to Refine AI-Generative Art Towards Precise Expressions. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–29.

[67] Zhijun Wang, Xuebo Liu, and Min Zhang. 2022. Breaking the representation bottleneck of chinese characters: Neural machine translation with stroke sequence modeling. *arXiv preprint arXiv:2211.12781* (2022).

[68] Albert Webson and Ellie Pavlick. 2021. Do prompt-based models really understand the meaning of their prompts? *arXiv preprint arXiv:2109.01247* (2021).

[69] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[70] Lauren Wilcox, Jie Lu, Jennifer Lai, Steven Feiner, and Desmond Jordan. 2009. ActiveNotes: computer-assisted creation of patient progress notes. In *CHI'09 Extended Abstracts on Human Factors in Computing Systems*. 3323–3328.

[71] Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. 2022. Promptchainer: Chaining large language model prompts through visual programming. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. 1–10.

[72] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts. In *Proceedings of the 2022 CHI conference on human factors in computing systems*. 1–22.

[73] Xingjiao Wu, Luwei Xiao, Yixuan Sun, Junhang Zhang, Tianlong Ma, and Liang He. 2022. A Survey of Human-in-the-Loop for Machine Learning. *Future Gener. Comput. Syst.* 135, C (oct 2022), 364–381. https://doi.org/10.1016/j.future.2022.05.014

[74] Xi Ye and Greg Durrett. 2022. The unreliability of explanations in few-shot prompting for textual reasoning. *Advances in neural information processing systems* 35 (2022), 30378–30392.

[75] Jeffrey M Zacks and Barbara Tversky. 2001. Event structure in perception and conception. *Psychological bulletin* 127, 1 (2001), 3.

[76] Haoqi Zhang, Edith Law, Rob Miller, Krzysztof Gajos, David Parkes, and Eric Horvitz. 2012. Human computation tasks with global constraints. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 217–226.

[77] Yiming Zhang, Shi Feng, and Chenhao Tan. 2022. Active example selection for in-context learning. *arXiv preprint arXiv:2211.04486* (2022).

[78] Yi Zhang, Sujay Kumar Jauhar, Julia Kiseleva, Ryen White, and Dan Roth. 2021. Learning to decompose and organize complex tasks. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2726–2735.

[79] Zheng Zhang, Ying Xu, Yanhao Wang, Bingsheng Yao, Daniel Ritchie, Tongshuang Wu, Mo Yu, Dakuo Wang, and Toby Jia-Jun Li. 2022. Storybuddy: A human-ai collaborative chatbot for parent-child interactive storytelling with flexible parental involvement. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–21.

[80] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493* (2022).

[81] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In *International Conference on Machine Learning*. PMLR, 12697–12706.

[82] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625* (2022).

[83] Tianshu Zhou and Jingsong Li. 2017. An Intelligent Writing Assistant Module for Narrative Clinical Records based on Named Entity Recognition and Similarity Computation. (2017).

[84] Kangli Zi, Shi Wang, Yu Liu, Jicun Li, Yanan Cao, and Cungen Cao. 2021. SOM-NCSCM: An Efficient Neural Chinese Sentence Compression Model Enhanced with Self-Organizing Map. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 403–415.

[85] Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. 2019. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593* (2019).

# A PROMPTS USED IN THE UPDATE STEPS

## A.1 Description of the LAN

=== LAN description starts ===

The task of the LAN is <task description>.

It is composed of the following agents:

1. <agent name>: <agent task description>.

2. ...

Here is the data flow among the agents:

1. from <source agent name> to <destination agent name>: <data content>.

2. ...

The input of the LAN is <LAN input>.

The output of the LAN is <LAN output>.

=== LAN description ends ===

### A.2 Description of the agent

=== Agent (<agent name>) description starts ===
Name: <agent name>.
Subtask: <agent task description>.
Output requirement: <output description>.
CM knowledge:
1. <CM knowledge 1>.
2. ...
CM examples:
1. <CM example 1>.
2. ...
EM knowledge:
1. <EM knowledge 1>.
2. ...
EM examples:
1. <EM example 1>.
2. ...
Inputs:
1. <input agent name>: <input agent task>. Its result: <input agent result>.
2. ...
Thought process: <agent thought process>.
Output: <agent output>.
=== Agent (<agent name>) description starts ===

### A.3 Step 1

Here is the description of the LAN:
// the description contains the input and output of the LAN
<description of the LAN>
Here is the ground truth: <ground truth>
Your task is the gap between the LAN's output and the ground truth.
You should first think step by step and output your reasoning process.
You should finally generate your result according to the format:
{"gap": ...}
The value of "gap" is a string, indicating the gap you find between the LAN output and the ground truth.

### A.4 Step 2

Here is the description of the LAN:
<description of the LAN>
Here is the ground truth: <ground truth>
The gap between the LAN's output and the ground truth: <gap>
Your task is to find out why the gap exists.
You should first think step by step and output your reasoning process.
You should finally generate your result according to the format:
{"reason_type": ..., "agent_name": ..., "reason_content": ...}
The value of "reason_type" should be one of "lack_of_agent", "poor_performance_of_agent", or "unexpected_execution_of_agent".
// detailed explanation of the template
If you choose "lack_of_agent":
- It means the gap exists because the LAN lacks an important agent.
- The "agent_name" should be the name of the absent agent.
- The "reason_content" should be why you think the LAN should have such an agent.
If you choose "poor_performance_of_agent":
- It means the gap exists because an agent in the LAN can not provide a satisfactory result.
- The "agent_name" should be the name of the agent that performs badly.
- The "reason_content" should be a description of the bad performance and why you think the agent performs badly.
If you choose "unexpected_execution_of_agent":
- It means the gap exists because an agent that should not execute executes.
- The "agent_name" should be the name of the agent that should not execute.
- The "reason_content" should be why you think the agent should not execute.

### A.5 Step 3

// the system enters Step 4 only when "reason_type" == "poor_performance_of_agent"
Here is the agent that causes the bad performance of the LAN:
<description of the agent>
Here is the information about the agents in the LAN:
1. <agent name>: <agent task description>.
2. ...
Here is the description of the bad performance:
// the description comes from the result of the last step
<description of the bad performance>
Your task is to find out why the agent has a bad performance.
You should first think step by step and output your reasoning process.
You should finally generate your result according to the format:
{"reason_type": ..., "reason_content": xxx}
// detailed explanation of the template

The value of "reason_type" should be one of "bad_structure", "lack_of_knowledge", "lack_of_input", or "not_executed".
If you choose "bad_structure" for "reason_type":
- It means the agent's task is too complex, and it should be split into multiple agents to improve its performance.
If you choose "lack_of_knowledge":
- It means the agent requires additional knowledge to improve its performance.
If you choose "lack_of_input":
- It means the agent requires additional inputs from other agents to improve its performance.
If you choose "not_executed":
- It means the agent does not execute because of an error in its CM.
You should indicate why you make such a decision in "reason_content".

### A.6 Step 4

EasyLAN uses different prompts according to different causes of the errors.

#### *A.6.1 lack_of_agent.*

The LAN has a bad performance because it lacks an important agent.
Here is the description of the LAN:
<description of the LAN>
The expected ground truth is <ground truth>
The reason why the LAN requires an additional agent is:
<reason content (from step 2)>
Your task is to calculate the parameters of the agent to be added to the LAN.
You should first think step by step and output your reasoning process.
You should finally generate your result according to the format:
{"parameters": {"agent_name": ..., "agent_task": ..., "output_desc": ...,
"CM_enabled": <true/false>, "CM_knowledge": [...], "EM_knowledge": [...],
"predecessors": [...], "descendants": [...]}}
// detailed explanation of the template
The properties mean:
- agent_name: the name of the new agent.
- agent_task: the subtask of the new agent.
- output_desc: the output description of the new agent.
- CM_enabled: choose from "true" or "false". "false" means that the agent will execute whatever its inputs are.
- CM_knowledge: knowledge that helps determine whether the agent should execute or not.
- EM_knowledge: knowledge that helps the agent execute.
- predecessors: a list of agent names. The agent will receive their results as input.
- descendants: a list of agent names. The result of the agent will be their input.

#### *A.6.2 unexpected_execution_of_agent.*

The LAN has a bad performance because an agent unexpectedly executes.

Here is the description of the LAN:
<description of the LAN>
The expected ground truth is <ground truth>
Here are the details about the agent that should not execute:
<description of the agent>
The reason why the agent should not execute is:
<reason content (from step 2)>
Your task is to add knowledge to the CM of the agent to deactivate it.
You should first think step by step and output your reasoning process.
You should finally generate your result according to the format:
{"parameters": {"new_CM_knowledge": [...]}}
// detailed explanation of the template
The properties mean:
- new_CM_knowledge: the knowledge that should be added to the CM of the agent.

*A.6.3 bad_structure.*
The LAN has a bad performance because an agent has a bad performance.
Here is the description of the LAN:
<description of the LAN>
The expected ground truth is <ground truth>
Here are the details about the agent that has a bad performance:
<description of the agent>
The agent should be split into multiple agents because:
<reason content (from step 3)>
Your task is to calculate the parameters of the split result.
You should first think step by step and output your reasoning process.
You should finally generate your result according to the format:
{"parameters": [{"agent_name": ..., "agent_task": ..., "output_desc": ...,
"CM_enabled": <true/false>, "CM_knowledge": [...], "EM_knowledge": ...,
"predecessors": [...], "descendants": [...]}, ...]}
// detailed explanation of the template
The "parameters" should be a list of objects. Each object has the following properties:
- agent_name: the name of the new agent.
- agent_task: the subtask of the new agent.
- output_desc: the output description of the new agent.
- CM_enabled: choose from "true" or "false". "false" means that the agent will execute whatever its inputs are.
- CM_knowledge: knowledge that helps determine whether the agent should execute or not.
- EM_knowledge: knowledge that helps the agent execute.
- predecessors: a list of agent names. The agent will receive their results as input.

- descendants: a list of agent names. The result of the agent will be their input.

#### *A.6.4 lack_of_knowledge.*

The LAN has a bad performance because an agent has a bad performance.
Here is the description of the LAN:
<description of the LAN>
The expected ground truth is <ground truth>
Here are the details about the agent that has a bad performance:
<description of the agent>
The agent requires additional knowledge to improve its performance because:
<reason content (from step 3)>
Your task is to calculate the additional knowledge for the agent.
You should first think step by step and output your reasoning process.
You should finally generate your result according to the format:
{"parameters": {"new_EM_knowledge": [...]}}
// detailed explanation of the template
The properties mean:
- new_EM_knowledge: the knowledge that should be added to the EM of the agent.

#### *A.6.5 lack_of_input.*

The LAN has a bad performance because an agent has a bad performance.
Here is the description of the LAN:
<description of the LAN>
The expected ground truth is <ground truth>
Here are the details about the agent that has a bad performance:
<description of the agent>
The agent requires additional input to improve its performance because:
<reason content (from step 3)>
Your task is to calculate the source of the additional inputs for the agent.
You should first think step by step and output your reasoning process.
You should finally generate your result according to the format:
{"parameters": {"new_predecessors": [...]}}
// detailed explanation of the template
The properties mean:
- new_predecessors: the names of the agents from which the agent should receive their results.

#### *A.6.6 not_executed.*

The LAN has a bad performance because an agent has a bad performance.
Here is the description of the LAN:

<description of the LAN>
The expected ground truth is <ground truth>
Here are the details about the agent that has a bad performance:
<description of the agent>
The agent does not execute. However, it should execute because:
<reason content (from step 3)>
Your task is to add knowledge to the CM of the agent to activate it.
You should first think step by step and output your reasoning process.
You should finally generate your result according to the format:
{"parameters": {"new_CM_knowledge": [...]}}
// detailed explanation of the template
The properties mean:
- new_CM_knowledge: the knowledge that should be added to the CM of the agent.

## B DETAILS ABOUT THE TASKS IN THE EVALUATION STUDY

### B.1 Tutorial task

The constructed LAN needs to be able to accurately identify errors within sentences. There are four types of errors present in both the training and test examples: (1) subject-verb agreement errors, (2) missing sentence components, (3) sentence redundancy, and (4) sentence ambiguity. An input sentence may contain multiple errors. It should be noted that this task may be relatively straightforward for an LLM. Our primary objective is to demonstrate how to utilize EasyLAN effectively.

**Example 1**

Input: The dogs runs quickly and fast.
Output: (1) Subject-Verb Agreement Error: The subject "dogs" is plural, but the verb "runs" is singular. (2) Redundancy: The sentence uses both "quickly" and "fast" to describe speed, which is redundant.

**Example 2**

Input: When John spoke to Bob, he angry.
Output: (1) Missing Component: The verb is missing for the subject "he" in the latter part of the sentence. (2) Ambiguity: The pronoun "he" is ambiguous because it's unclear whether "he" refers to John or Bob.

### B.2 Task 1

The constructed LAN needs to be capable of translating Classical Chinese into English. There are four requirements for the translation result:

(1) The result is a grammatically correct sentence.
(2) The result should correctly choose whether to rhyme or not. If the input is a rhyming verse, the output should also preserve the rhyme scheme.
(3) The result should accurately translate words with multiple meanings.

(4) The result should correctly interpret metaphors specific to the Chinese context.

**Example 1**

Input: 古 (Ancient) 之 (Of) 学 (Learn) 者 (Person) 必 (Must) 有 (Have) 师 (Teacher)

Explain: The words in the parentheses are literal translations of the corresponding Chinese characters. Although another meaning of "学者" is "scholar," it doesn't fit the context here.

Output: In ancient times, those who wished to learn would surely have a teacher.

**Example 2**

Input: 春 (Spring) 眠 (Sleep) 不 (Not) 觉 (Aware) 晓 (Dawn)，处 (Place) 处 (Place) 闻 (Hear) 啼 (Chirp) 鸟 (Bird)

Explain: "晓" sounds like /shiao/, and "鸟" sounds like /niao/. They both end with the same vowel, making them rhyme.

Output: This morning of spring in bed I'm lying, Not wake up till I hear birds crying.

**Example 3**

Input: 居 (Reside) 庙 (Temple) 堂 (Hall) 之 (Of) 上 (Above) 则 (Then) 忧 (Worry) 其 (Their) 民 (People)

Explain: In Chinese, "庙堂" is a metaphorical term referring to the government.

Output: In high offices, they were concerned with their subject citizens.

### B.3 Task 2

The constructed LAN can abbreviate a Chinese sentence while retaining only its subject, predicate, and object. There are four requirements for the result:

(1) The result is a grammatically correct sentence.
(2) The result should remove all the attributives.
(3) The result should remove all the adverbials.
(4) The result should remove all the complements.

**Example**

Input: 小 (Little) 鸟 (Bird) 奋力地 (Vigorously) 拍打 (Flap) 翅膀 (Wings) 以 (To) 飞得 (Fly) 更高 (Higher)

Explain: (1) "小 (Little)" should be removed because it is an attributive of "鸟 (Bird)". (2) "奋力地 (Vigorously)" should be removed because it is an adverbial of "拍打 (Flap)". (3) "以 (To) 飞得 (Fly) 更高 (Higher)" should be removed because it is a complement that describes the goal of "拍打 (Flap)".

Output: 鸟 (Bird) 拍打 (Flap) 翅膀 (Wings)

### B.4 Task 3

The constructed LAN can achieve: generate a lower couplet based on a given Chinese upper couplet (also called duilian[3]). There are four requirements for the result:

(1) The upper and lower couplets have an equal number of characters.

[3]The example for this task is adapted from the following web page: https://en.wikipedia.org/wiki/Duilian_(poetry)

(2) The upper and lower couplets exhibit similar emotions and closely related themes.
(3) Corresponding words in the upper and lower couplets share the same parts of speech.
(4) In the upper couplet, when two positions share the same character, the corresponding positions in the lower couplet must also share the same character, and vice versa.

**Example**

Input: 书 (book) 山 (mountain) 有 (have) 路 (way) 勤 (diligence) 为 (is) 径 (path)
Output: 学 (knowledge) 海 (see) 无 (have not) 涯 (border) 苦 (painstaking) 作 (make) 舟 (boat)
Explain:

(1) Both the upper and lower couplets consist of 7 characters each.
(2) Both the upper and lower couplets serve as reminders for people to study diligently.
(3) Corresponding words share the same part of speech. For example, the second-to-last word ('为 is' and '作 make') in both couplets are verbs, while the last word ('径 path' and '舟 boat') in both couplets are nouns.
(4) In the upper couplet, there are no repeated characters, and therefore, there are no repeated characters in the lower couplet as well.