EHRAgent: Code Empowers Large Language Models for Few-shot Complex Tabular Reasoning on Electronic Health Records

Wenqi Shi[1*]   **Ran Xu**[2*]   **Yuchen Zhuang**[1]   **Yue Yu**[1]   **Jieyu Zhang**[3]
Hang Wu[1]   **Yuanda Zhu**[1]   **Joyce Ho**[2]   **Carl Yang**[2]   **May D. Wang**[1]

[1] Georgia Institute of Technology   [2] Emory University   [3] University of Washington

{wqshi,yczhuang,yueyu,hangwu,yzhu94,maywang}@gatech.edu,
{ran.xu,joyce.c.ho,j.carlyang}@emory.edu, jieyuz2@cs.washington.edu

Abstract

Clinicians often rely on data engineers to retrieve complex patient information from electronic health record (EHR) systems, a process that is both inefficient and time-consuming. We propose EHRAgent[1], a large language model (LLM) agent empowered with accumulative domain knowledge and robust coding capability. EHRAgent enables autonomous code generation and execution to facilitate clinicians in directly interacting with EHRs using natural language. Specifically, we formulate a multi-tabular reasoning task based on EHRs as a tool-use planning process, efficiently decomposing a complex task into a sequence of manageable actions with external toolsets. We first inject relevant medical information to enable EHRAgent to effectively reason about the given query, identifying and extracting the required records from the appropriate tables. By integrating interactive coding and execution feedback, EHRAgent then effectively learns from error messages and iteratively improves its originally generated code. Experiments on three real-world EHR datasets show that EHRAgent outperforms the strongest baseline by up to 29.6% in success rate, verifying its strong capacity to tackle complex clinical tasks with minimal demonstrations.

1 Introduction

An electronic health record (EHR) is a digital version of a patient's medical history maintained by healthcare providers over time (Gunter and Terry, 2005). In clinical research and practice, clinicians actively interact with EHR systems to access and retrieve patient data, ranging from detailed individual-level records to comprehensive population-level insights (Cowie et al., 2017). The reliance on pre-defined rule-based conversion systems in most EHRs often necessitates additional training or assistance from data engineers for clinicians to obtain information beyond these rules (Mandel et al., 2016; Bender and Sartipi, 2013), leading to inefficiencies and delays that may impact the quality and timeliness of patient care.

Alternatively, an autonomous agent could facilitate clinicians to communicate with EHRs in natural languages, translating clinical questions into machine-interpretable queries, planning a sequence of actions, and ultimately delivering the final responses. Compared to existing EHR management that relies heavily on human effort, the adoption of autonomous agents holds great potential to efficiently simplify workflows and reduce workloads for clinicians (Figure 1). Although several supervised learning approaches (Lee et al., 2022; Wang et al., 2020) have been explored to automate the translation of clinical questions into corresponding machine queries, such systems require extensive training samples with fine-grained annotations, which are both expensive and challenging to obtain.

Large language models (LLMs) (OpenAI, 2023; Anil et al., 2023) bring us one step closer to autonomous agents with extensive knowledge and substantial instruction-following abilities from di-
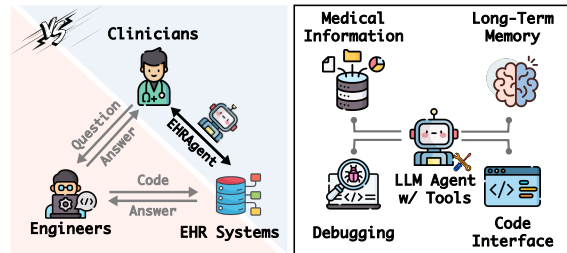

Figure 1: Simple and efficient interactions between clinicians and EHR systems with the assistance of LLM agents. Clinicians specify tasks in natural language, and the LLM agent autonomously generates and executes code to interact with EHRs (*right*) for answers. It eliminates the need for specialized expertise or extra effort from data engineers, which is typically required when dealing with EHRs in existing clinical settings (*left*).

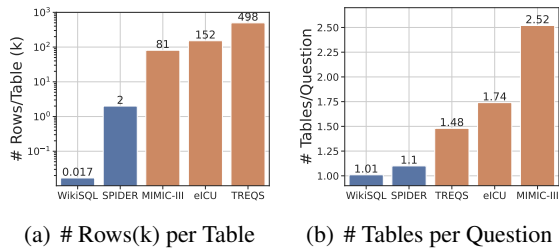(a) # Rows(k) per Table (b) # Tables per Question

Figure 2: Compared to general domain tasks (blue) such as WikiSQL (Zhong et al., 2017) and SPIDER (Yu et al., 2018), multi-tabular reasoning tasks within EHRs (orange) typically involve a significantly larger number of records per table and necessitate querying multiple tables to answer each question, thereby requiring more advanced reasoning and problem-solving capabilities.

verse corpora during pretraining. LLM-based autonomous agents have demonstrated remarkable capabilities in problem-solving, such as reasoning (Wei et al., 2022), planning (Yao et al., 2023b), and memorizing (Wang et al., 2023b). One particularly notable capability of LLM agents is tool-usage (Schick et al., 2023; Qin et al., 2023), where they can utilize external tools (*e.g.*, calculators, APIs, *etc.*), interact with environments, and generate action plans with intermediate reasoning steps that can be executed sequentially towards a valid solution (Wu et al., 2023; Zhang et al., 2023).

Despite their success in general domains, LLMs have encountered unique and significant challenges in the medical domain (Jiang et al., 2023; Yang et al., 2022; Moor et al., 2023), especially when dealing with individual EHR queries that require advanced reasoning across a vast number of records within multiple tables (Li et al., 2024; Lee et al., 2022) (Figure 2). First, given the constraints in both the volume and specificity of training data within the medical field (Thapa and Adhikari, 2023), LLMs still struggle to identify and extract relevant information from the appropriate tables and records within EHRs, due to insufficient knowledge and understanding of their complex structure and content. Second, EHRs are typically large-scale relational databases containing vast amounts of tables with comprehensive administrative and clinical information (*e.g.*, 26 tables of 46K patients in MIMIC-III). Moreover, real-world clinical tasks derived from individual patients or specific groups are highly diverse and complex, requiring multi-step or complicated operations.

To address these limitations, we propose EHRAgent, an autonomous LLM agent with external tools and code interface for improved multi-tabular reasoning across EHRs. We translate the EHR question-answering problem into a tool-use planning process – generating, executing, debugging, and optimizing a sequence of code-based actions. Firstly, to overcome the lack of domain knowledge in LLMs, we instruct EHRAgent to integrate query-specific medical information for effectively reasoning from the given query and locating the query-related tables or records. Moreover, we incorporate long-term memory to continuously maintain a set of successful cases and dynamically select the most relevant few-shot examples, in order to effectively learn from and improve upon past experiences. Secondly, we establish an interactive coding mechanism, which involves a multi-turn dialogue between the code planner and executor, iteratively refining the generated code-based plan for complex multi-hop reasoning. Specifically, EHRAgent optimizes the execution plan by incorporating environment feedback and delving into error messages to enhance debugging proficiency.

We conduct extensive experiments on three large-scale real-world EHR datasets to validate the empirical effectiveness of EHRAgent, with a particular focus on challenging tasks that reflect diverse information needs and align with real-world application scenarios. In contrast to traditional supervised settings (Lee et al., 2022; Wang et al., 2020) that require over 10K training samples with manually crafted annotations, EHRAgent demonstrates its efficiency by necessitating only four demonstrations. Our findings suggest that EHRAgent improves multi-tabular reasoning on EHRs through autonomous code generation and execution, leveraging accumulative domain knowledge and interactive environmental feedback.

Our main contributions are as follows:

• We propose EHRAgent, an LLM agent augmented with external tools and domain knowledge, to solve few-shot multi-tabular reasoning derived from EHRs with only four demonstrations;

• Planning with a code interface, EHRAgent formulates a complex clinical problem-solving process as an executable code plan of action sequences, along with a code executor;

• We introduce interactive coding between the LLM agent and code executor, iteratively refining plan generation and optimizing code execution by examining environmental feedback in depth;

• Experiments on three EHR datasets show that EHRAgent improves the strongest baseline on multi-hop reasoning by up to 29.6% in success rate.
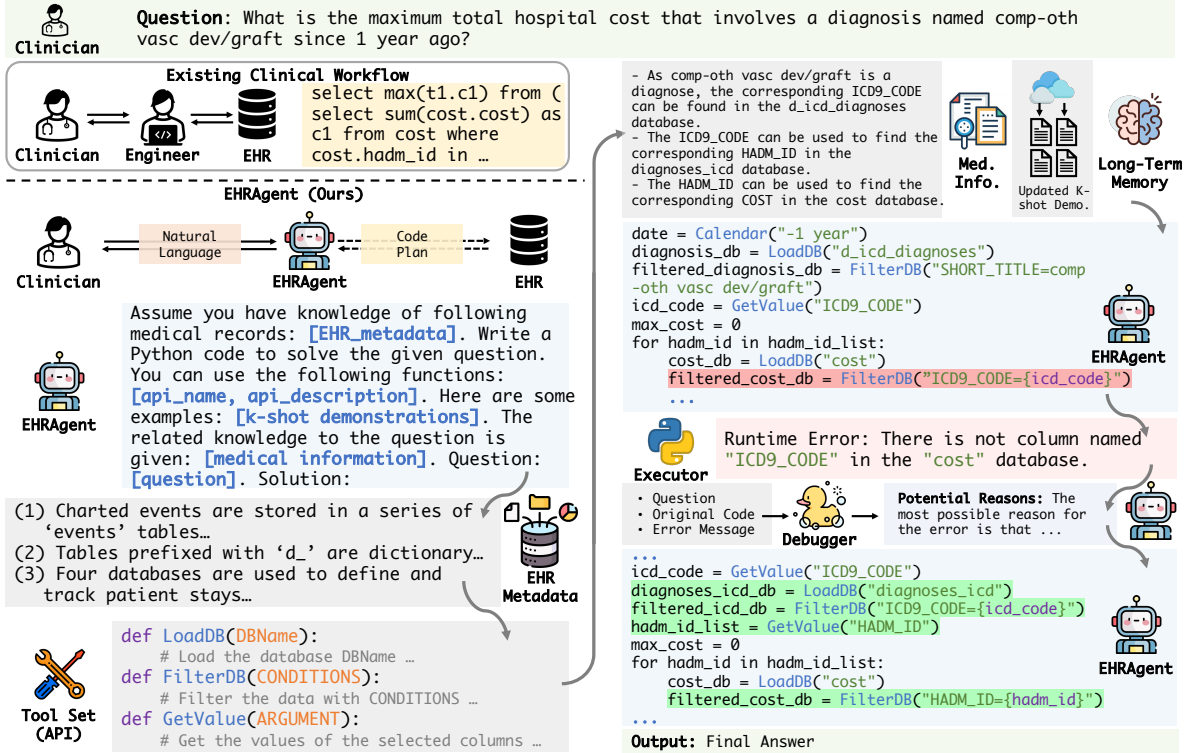
Figure 3: Overview of our proposed LLM agent, EHRAgent, for complex few-shot tabular reasoning tasks on EHRs. Given an input clinical question based on EHRs, EHRAgent decomposes the task and generates a plan (*i.e.*, code) based on (a) metadata (*i.e.*, descriptions of tables and columns in EHRs), (b) tool function definitions, (c) few-shot examples, and (d) domain knowledge (*i.e.*, integrated medical information). Upon execution, EHRAgent iteratively debugs the generated code following the execution errors and ultimately generates the final solution.

2 Preliminaries

Problem Formulation. In this work, we focus on addressing health-related queries by leveraging information from structured EHRs. The reference EHR, denoted as $\mathcal{R} = \{R_0, R_1, \cdots\}$, comprises multiple tables, while $\mathcal{C} = \{C_0, C_1, \cdots\}$ corresponds to the column descriptions within $\mathcal{R}$. For each given query in natural language, denoted as q, our goal is to extract the final answer by utilizing the information within both $\mathcal{R}$ and $\mathcal{C}$.

LLM Agent Setup. We further formulate the planning process for LLMs as autonomous agents in EHR question answering. For initialization, the LLM agent is equipped with a set of pre-built tools $\mathcal{M} = \{M_0, M_1, \cdots\}$ to interact with and address queries derived from EHRs $\mathcal{R}$. Given an input query $q \in \mathcal{Q}$ from the task space $\mathcal{Q}$, the objective of the LLM agent is to design a T-step execution plan $P = (a_1, a_2, \cdots, a_T)$, with each action a_t selected from the tool set $a_t \in \mathcal{M}$. Specifically, we generate the action sequences (*i.e.*, plan) by prompting the LLM agent following a policy $p_q \sim \pi(a_1, \cdots, a_{T_q} | q; \mathcal{R}, \mathcal{M}) : \mathcal{Q} \times \mathcal{R} \times \mathcal{M} \rightarrow \Delta(\mathcal{M})^{T_q}$, where $\Delta(\cdot)$ is a probability simplex func-

tion. The final output is obtained by executing the entire plan $y \sim \rho(y | q, a_1, \cdots, a_{T_q})$, where ρ is a plan executor interacting with EHRs.

Planning with Code Interface. To mitigate ambiguities and misinterpretations in plan generation, an increasing number of LLM agents (Gao et al., 2023; Liang et al., 2023; Sun et al., 2023; Chen et al., 2023; Zhuang et al., 2024) employ code prompts as planner interface instead of natural language prompts. The code interface enables LLM agents to formulate an executable code plan as action sequences, intuitively transforming natural language question-answering into iterative coding (Yang et al., 2023). Consequently, the planning policy $\pi(\cdot)$ turns into a code generation process, with a code execution as the executor $\rho(\cdot)$. We then track the outcome of each interaction back to the LLM agent, which can be either a successful execution result or an error message, to iteratively refine the generated code-based plan. This interactive process, a multi-turn dialogue between the planner and executor, takes advantage of the advanced reasoning capabilities of LLMs to optimize plan refinement and execution.

Algorithm 1: Overview of EHRAgent.

Input: q: input question; $\mathcal{R}$: reference EHRs; $\mathcal{C}_i$: column description of EHR R_i; $\mathcal{D}$: descriptions of EHRs $\mathcal{R}$; T: the maximum number of steps; $\mathcal{T}$: definitions of tool function; $\mathcal{L}$: long-term memory.

Initialize $t \leftarrow 0$, $C^{(0)}(q) \leftarrow \varnothing$, $O^{(0)}(q) \leftarrow \varnothing$

// Medical Information Integration
$\mathcal{I} = [\mathcal{D}; \mathcal{C}_0; \mathcal{C}_1; \cdots]$
$B(q) = \text{LLM}([\mathcal{I}; q])$

// Examples Retrieval from Long-Term Memory
$\mathcal{E}(q) = \arg \text{TopK}_{\max}(\text{sim}(q, q_i | q_i \in \mathcal{L}))$

// Plan Generation
$C^{(0)}(q) = \text{LLM}([\mathcal{I}; \mathcal{T}; \mathcal{E}(q); q; B(q)])$

while $t < T$ & TERMINATE $\notin O^{(t)}(q)$ **do**

> *// Code Execution*
> $O^{(t)}(q) = \text{EXECUTE}(C^{(t)}(q))$
> *// Debugging and Plan Modification*
> $C^{(t+1)}(q) = \text{LLM}(\text{DEBUG}(O^{(t)}(q)))$
> $t \leftarrow t + 1$

Output: Final answer (solved) or error message (unsolved) from $O^{(t)}(q)$.

---

3 EHRAgent: LLMs as Medical Agents

In this section, we present EHRAgent (Figure 3), an LLM agent that enables multi-turn interactive coding to address multi-hop reasoning tasks on EHRs. EHRAgent comprises four key components: (1) **Medical Information Integration:** We incorporate query-specific medical information for effective reasoning based on the given query, enabling EHRAgent to identify and retrieve the necessary tables and records for answering the question. (2) **Demonstration Optimization through Long-Term Memory:** Using long-term memory, EHRAgent replaces original few-shot demonstrations with the most relevant successful cases retrieved from past experiences. (3) **Interactive Coding with Execution Feedback:** EHRAgent harnesses LLMs as autonomous agents in a multi-turn conversation with a code executor. (4) **Rubber Duck Debugging via Error Tracing:** Rather than simply sending back information from the code executor, EHRAgent thoroughly analyzes error messages to identify the underlying causes of errors through iterations until a final solution. We summarize the workflow of EHRAgent in Algorithm 1.

3.1 Medical Information Integration

Clinicians frequently pose complex inquiries that necessitate advanced reasoning across multiple tables and access to a vast number of records within a single query. To accurately identify the required tables, we first incorporate query-specific medical information (*i.e.*, domain knowledge) into EHRAgent

to develop a comprehensive understanding of the query within a limited context length. Given an EHR-based clinical question q and the reference EHRs $\mathcal{R} = \{R_0, R_1, \cdots\}$, the objective of information integration is to generate the domain knowledge most relevant to q, thereby facilitating the identification and location of potential useful references within $\mathcal{R}$. For example, given a query related to '*Aspirin*', we expect LLMs to locate the drug '*Aspirin*' at the PRESCRIPTION table, under the *prescription_name* column in the EHR.

To achieve this, we initially maintain a thorough metadata $\mathcal{I}$ of all the reference EHRs, including overall data descriptions $\mathcal{D}$ and the detailed column descriptions $\mathcal{C}_i$ for each individual EHR R_i, expressed as $\mathcal{I} = [\mathcal{D}; \mathcal{C}_0; \mathcal{C}_1; \cdots]$. To further extract additional background knowledge essential for addressing the complex query q, we then distill key information from the detailed introduction $\mathcal{I}$. Specifically, we directly prompt LLMs to generate the relevant information $B(q)$ based on demonstrations, denoted as $B(q) = \text{LLM}([\mathcal{I}; q])$.

3.2 Demonstration Optimization through Long-Term Memory

Due to the vast volume of information within EHRs and the complexity of the clinical questions, there exists a conflict between limited input context length and the number of few-shot examples. Specifically, K-shot examples may not adequately cover the entire question types as well as the EHR information. To address this, we maintain a long-term memory $\mathcal{L}$ for storing past successful code snippets and reorganizing few-shot examples by retrieving the most relevant samples from $\mathcal{L}$. Consequently, the LLM agent can learn from and apply patterns observed in past successes to current queries. The selection of K-shot demonstrations $\mathcal{E}(q)$ is defined as follows:

$$\mathcal{E}(q) = \arg \text{TopK}_{\max}(\text{sim}(q, q_i | q_i \in \mathcal{L})), \quad (1)$$

where $\arg \text{TopK} \max(\cdot)$ identifies the indices of the top K elements with the highest values from $\mathcal{L}$, and $\text{sim}(\cdot, \cdot)$ calculates the similarity between two questions, employing negative Levenshtein distance as the similarity metric. Following this retrieval process, the newly acquired K-shot examples $\mathcal{E}(q)$ replace the originally predefined examples $\mathcal{E} = \{E_1, \cdots, E_K\}$. This updated set of examples serves to reformulate the prompt, guiding EHRAgent in optimal demonstration selection by leveraging accumulative domain knowledge.

3.3 Interactive Coding with Execution

We then introduce interactive coding between the LLM agent (*i.e.*, code generator) and code executor to facilitate iterative plan refinement. EHRAgent integrates LLMs with a code executor in a multi-turn conversation. The code executor runs the generated code and returns the results to the LLM. Within the conversation, EHRAgent navigates the subsequent phase of the dialogue, where the LLM agent is expected to either (1) continue to iteratively refine its original code in response to any errors encountered or (2) finally deliver a conclusive answer based on the successful execution outcomes.

LLM Agent. To generate accurate code snippets $C(q)$ as solution plans for the query q, we prompt the LLM agent with a combination of the EHR introduction $\mathcal{I}$, tool function definitions $\mathcal{T}$, a set of K-shot examples $\mathcal{E}(q)$ updated by long-term memory, the input query q, and the integrated medical information relevant to the query $B(q)$:

$$C(q) = \text{LLM}([\mathcal{I}; \mathcal{T}; \mathcal{E}(q); q; B(q)]). \quad (2)$$

We develop the LLM agent to (1) generate code within a designated coding block as required, (2) modify the code according to the outcomes of its execution, and (3) insert a specific code âĂĲTER-MINATEâĂİ at the end of its response to indicate the conclusion of the conversation.

Code Executor. The code executor automatically extracts the code from the LLM agent's output and executes it within the local environment: $O(q) = \text{EXECUTE}(C(q))$. After execution, it sends back the execution results to the LLM agent for potential plan refinement and further processing. Given the alignment of empirical observations and Python's inherent modularity with tool functions[2], we select Python 3.9 as the primary coding language for interactions between the LLM agent and the code executor.

3.4 Rubber Duck Debugging via Error Tracing

Our empirical observations indicate that LLM agents tend to make slight modifications to the code snippets based on the error message without further debugging. In contrast, human programmers often delve deeper, identifying bugs or underlying causes by analyzing the code implementation against the error descriptions (Chen et al., 2024). Inspired

---

[2]We include additional analysis in Appendix D to further justify the selection of primary programming language.

by this, we integrate a 'rubber duck debugging' pipeline with error tracing to refine plans with the LLM agent. Specifically, we provide detailed trace feedback, including error type, message, and location, all parsed from the error information by the code executor. Subsequently, this error context is presented to a 'rubber duck' LLM, prompting it to generate the most probable causes of the error. The generated explanations are then fed back into the conversation flow, aiding in the debugging process. For the t-th interaction between the LLM agent and the code executor, the process is as follows:

$$
\begin{aligned}
O^{(t)}(q) &= \text{EXECUTE}(C^{(t)}(q)), \\
C^{(t+1)}(q) &= \text{LLM}(\text{DEBUG}(O^{(t)}(q))).
\end{aligned}
\quad (3)
$$

The interaction ends either when a 'TERMINATE' signal appears in the generated messages or when t reaches a pre-defined threshold of steps T.

4 Experiments

4.1 Experiment Setup

Tasks and Datasets. We evaluate EHRAgent on three publicly available structured EHR datasets, MIMIC-III (Johnson et al., 2016), eICU (Pollard et al., 2018), and TREQS (Wang et al., 2020) for multi-hop question and answering on EHRs. These questions originate from real-world clinical needs and cover a wide range of tabular queries commonly posed within EHRs. Our final dataset includes an average of 10.7 tables and 718.7 examples per dataset, with an average of 1.91 tables required to answer each question. We include additional dataset details in Appendix A.

Tool Sets. To enable LLMs in complex operations such as calculations and information retrieval, we integrate external tools in EHRAgent during the interaction with EHRs. Our toolkit can be easily expanded with natural language tool function definitions in a plug-and-play manner. Toolset details are available in Appendix B.

Baselines. We compare EHRAgent with nine LLM-based planning, tool use, and coding methods, including five baselines with natural language interfaces and four with coding interfaces. For a fair comparison, all baselines, including EHRAgent, utilize the same (a) EHR metadata, (b) tool definitions, and (c) initial few-shot demonstrations in the prompts by default. We summarize their implementations in Appendix C.

Evaluation Protocol. Following Yao et al. (2023b); Sun et al. (2023); Shinn et al. (2023), our

Dataset (→)	MIMIC-III						eICU					TREQS				
Complexity Level (→)	I	II	III	IV	All		I	II	III	All		I	II	III	All	
Methods (↓) /Metrics (→)	SR.				SR.	CR.	SR.			SR.	CR.	SR.			SR.	CR.
w/o Code Interface																
CoT (Wei et al., 2022)	29.33	12.88	3.08	2.11	9.58	38.23	26.73	33.00	8.33	27.34	65.65	11.22	9.15	0.00	9.84	54.02
Self-Consistency (Wang et al., 2023d)	33.33	16.56	4.62	1.05	10.17	40.34	27.11	34.67	6.25	31.72	70.69	12.60	11.16	0.00	11.45	57.83
Chameleon (Lu et al., 2023)	38.67	14.11	4.62	4.21	12.77	42.76	31.09	34.68	16.67	35.06	83.41	13.58	12.72	4.55	12.25	60.34
ReAct (Yao et al., 2023b)	34.67	12.27	3.85	2.11	10.38	25.92	27.82	34.24	15.38	33.33	73.68	33.86	26.12	9.09	29.22	78.31
Reflexion (Shinn et al., 2023)	41.05	19.31	12.57	11.96	19.48	57.07	38.08	33.33	15.38	36.72	80.00	35.04	29.91	9.09	31.53	80.02
w/ Code Interface																
LLM2SQL (Nan et al., 2023)	23.68	10.64	6.98	4.83	13.10	44.83	20.48	25.13	12.50	23.28	51.72	39.61	36.43	12.73	37.89	79.22
DIN-SQL (Pourreza and Rafiei, 2023)	49.51	44.22	36.25	21.85	38.45	81.72	23.49	26.13	12.50	25.00	55.00	41.34	36.38	12.73	38.05	82.73
Self-Debugging (Chen et al., 2024)	50.00	46.93	30.12	27.61	39.05	71.24	32.53	21.86	25.00	30.52	66.90	43.54	36.65	18.18	40.10	84.44
AutoGen (Wu et al., 2023)	36.00	28.13	15.33	11.11	22.49	61.47	42.77	40.70	18.75	40.69	86.21	46.65	19.42	0.00	33.13	85.38
EHRAgent (Ours)	**71.58**	**66.34**	**49.70**	**49.14**	**58.97**	**85.86**	**54.82**	**53.52**	**25.00**	**53.10**	**91.72**	**78.94**	**61.16**	**27.27**	**69.70**	**88.02**

Table 1: Main results of success rate (*i.e.*, SR.) and completion rate (*i.e.*, CR.) on MIMIC-III, eICU, and TREQS datasets. The complexity of questions increases from Level I (the simplest) to Level IV (the most difficult).

primary evaluation metric is *success rate*, quantifying the percentage of queries the model handles successfully. Following Xu et al. (2023); Kirk et al. (2024), we further assess *completion rate*, which represents the percentage of queries that the model can generate executable plans (even not yield correct results). We categorize input queries into complexity levels (I-IV) based on the number of tables involved in solution generation. We include more details in Appendix A.2.

Implementation Details. We employ GPT-4 (OpenAI, 2023) (version gpt-4-0613) as the base LLM model for all experiments. We set the temperature to 0 when making API calls to GPT-4 to eliminate randomness and set the pre-defined threshold of steps (T) to 10. Due to the maximum length limitations of input context in baselines (*e.g.*, ReAct and Chameleon), we use the same initial four-shot demonstrations ($K = 4$) for all baselines and EHRAgent to ensure a fair comparison. Appendix E provides additional implementation details with prompt templates.

4.2 Main Results

Table 1 summarizes the experimental results of EHRAgent and baselines on multi-tabular reasoning within EHRs. From the results, we have the following observations:

(1) EHRAgent significantly outperforms all the baselines on all three datasets with a performance gain of 19.92%, 12.41%, and 29.60%, respectively. This indicates the efficacy of our key designs, namely interactive coding with environment feedback and domain knowledge injection, as they gradually refine the generated code and provide sufficient background information during the planning process. Experimental results with additional base LLMs are available in Appendix F.1.

(2) *CoT*, *Self-Consistency*, and *Chameleon* all neglect environmental feedback and cannot adaptively refine their planning processes. Such deficiencies hinder their performance in EHR question-answering scenarios, as the success rates for these methods on three datasets are all below 40%.

(3) *ReAct* and *Reflexion* both consider environment feedback but are restricted to tool-generated error messages. Thus, they potentially overlook the overall planning process. Moreover, they both lack a code interface, which prevents them from efficient action planning, and results in lengthy context execution and lower completion rates.

(4) *LLM2SQL* and *DIN-SQL* leverage LLM to directly generate SQL queries for EHR question-answering tasks. However, the gain is rather limited, as the LLM still struggles to generate high-quality SQL codes for execution. Besides, the absence of the debugging module further impedes its overall performance on this challenging task.

(5) *Self-Debugging* and *AutoGen* present a notable performance gain over other baselines, as they leverage code interfaces and consider the errors from the coding environment, leading to a large improvement in the completion rate. However, as they fail to model medical knowledge or identify underlying causes from error patterns, their success rates are still sub-optimal.

4.3 Ablation Studies

Our ablation studies on MIMIC-III (Table 2) demonstrate the effectiveness of all four compo-

Complexity Level ($\rightarrow$)	I	II	III	IV	All	
Methods ($\downarrow$) /Metrics ($\rightarrow$)		SR.			SR.	CR.
EHRAgent	**71.58**	**66.34**	**49.70**	**49.14**	**58.97**	**85.86**
w/o medical information	68.42	33.33	29.63	20.00	33.66	69.22
w/o long-term memory	65.96	54.46	37.13	42.74	51.73	83.42
w/o interactive coding	45.33	23.90	20.97	13.33	24.55	62.14
w/o rubber duck debugging	55.00	38.46	41.67	35.71	42.86	77.19

Table 2: Ablation studies on success rate (*i.e.*, SR.) and completion rate (*i.e.*, CR.) under different question complexity (I-IV) on MIMIC-III dataset.

(a) success rate   (b) completion rate

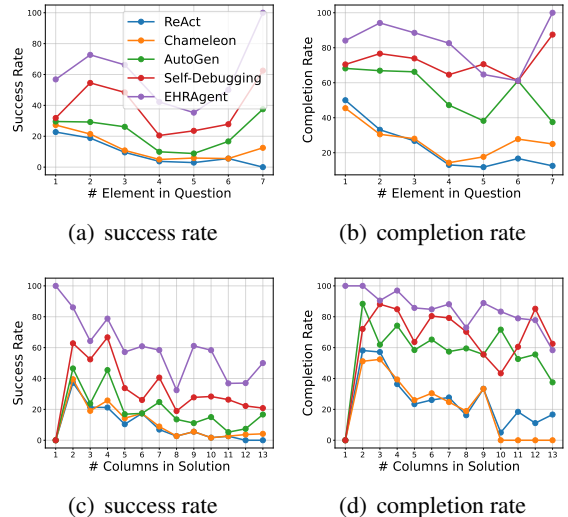(c) success rate   (d) completion rate

Figure 4: Success rate and completion rate under different question complexity, measured by the number of elements (*i.e.*, slots) in each question (*upper*) and the number of columns involved in each solution (*bottom*).

(a) success rate   (b) completion rate

Figure 5: Success rate and completion rate under different numbers of demonstrations.

nents in EHRAgent. Interactive coding[3] is the most significant contributor across all complexity levels, which highlights the importance of code generation in planning and environmental interaction for refinement. In addition, more challenging tasks benefits more from knowledge integration, indicating that comprehensive understanding of EHRs facilitates the complex multi-tabular reasoning in effective schema linking and reference (*e.g.*, tables, columns, and condition values) identification. Detailed analysis with additional settings and results is available in Appendix F.2.

4.4 Quantitative Analysis

Effect of Question Complexity. We take a closer look at the model performance by considering multi-dimensional measurements of question complexity, exhibited in Figure 4. Although the performances of both EHRAgent and the baselines generally decrease with an increase in task complexity (either quantified as more elements in queries or more columns in solutions), EHRAgent consistently outperforms all the baselines at various levels of difficulty. Appendix G.1 includes additional analysis on the effect of various question complexities.

Sample Efficiency. Figure 5 illustrates the model performance *w.r.t.* number of demonstrations for EHRAgent and the two strongest baselines, AutoGen and Self-Debugging. Compared to supervised learning like text-to-SQL (Wang et al., 2020; Raghavan et al., 2021; Lee et al., 2022) that requires extensive training on over 10K samples with detailed annotations (*e.g.*, manually generated corresponding code for each query), LLM agents enable complex tabular reasoning using a few demonstrations only. One interesting finding is that as the number of examples increases, both the success and completion rate of AutoGen tend to decrease,

mainly due to the context limitation of LLMs. Notably, the performance of EHRAgent remains stable with more demonstrations, which may benefit from its integration of a 'rubber duck' debugging module and the adaptive mechanism for selecting the most relevant demonstrations.

4.5 Error Analysis

Figure 6 presents a summary of error types identified in the solution generation process of EHRAgent based on the MIMIC-III, as determined through manual examinations and analysis. The majority of errors occur because the LLM agent consistently fails to identify the underlying cause of these errors within T-step trails, resulting in plans that are either incomplete or inexcusable. Additional analysis of each error type is available in Appendix G.2.

---

[3]For EHRAgent w/o interactive coding, we deteriorate from generating code-based to natural language-based plans and enable debugging based on error messages from tool execution.

Figure 6: Percentage of mistake examples in different categories on MIMIC-III dataset.

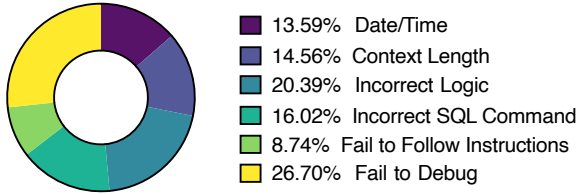Figure 7: Case study of EHRAgent harnessing LLMs in a multi-turn conversation with a code executor, debugging with execution errors through iterations.

4.6 Case Study

Figure 7 presents a case study of EHRAgent in interactive coding with environment feedback. The initial solution from LLM is unsatisfactory with multiple errors. Fortunately, EHRAgent is capable of identifying the underlying causes of errors by analyzing error messages and resolves multiple errors one by one through iterations. We have additional case studies in Appendix H.

5 Related Work

Augmenting LLMs with External Tools. LLMs have rapidly evolved from text generators into core computational engines of autonomous agents, with advanced planning and tool-use capabilities (Schick et al., 2023; Shen et al., 2023; Wang et al., 2024b; Yuan et al., 2024a,b; Zhuang et al.,

2023). LLM agents equip LLMs with planning capabilities (Yao et al., 2023a; Gong et al., 2023) to decompose a large and hard task into multiple smaller and simpler steps for efficiently navigating complex real-world scenarios. By integrating with external tools, LLM agents access external APIs for additional knowledge beyond training data (Lu et al., 2023; Patil et al., 2023; Qin et al., 2024; Li et al., 2023b,a). The disconnection between plan generation and execution, however, prevents LLM agents from effectively and efficiently mitigating error propagation and learning from environmental feedback (Qiao et al., 2023; Shinn et al., 2023; Yang et al., 2023). To this end, we leverage interactive coding to learn from dynamic interactions between the planner and executor, iteratively refining generated code by incorporating insights from error messages. Furthermore, EHRAgent extends beyond the limitation of short-term memory obtained from in-context learning, leveraging long-term memory (Sun et al., 2023; Zhang et al., 2023) by rapid retrieval of highly relevant and successful experiences accumulated over time.

LLM Agents for Scientific Discovery. Augmenting LLMs with domain-specific tools, LLM agents have demonstrated capabilities of autonomous design, planning, and execution in accelerating scientific discovery (Wang et al., 2023a,c, 2024a; Xi et al., 2023; Zhao et al., 2023; Cheung et al., 2024; Gao et al., 2024), including organic synthesis (Bran et al., 2023), material design (Boiko et al., 2023), and gene prioritization (Jin et al., 2024). In the medical field, MedAgents (Tang et al., 2023), a multi-agent collaboration framework, leverages role-playing LLM-based agents in a task-oriented multi-round discussion for multi-choice questions in medical entrance examinations. Similarly, Abbasian et al. (2023) develop a conversational agent to enhance LLMs using external tools for general medical question-answering tasks. Different from existing LLM agents in the medical domains that focus on improving tasks like multiple-choice question-answering, EHRAgent integrates LLMs with an interactive code interface, exploring complex few-shot tabular reasoning tasks derived from real-world EHRs through autonomous code generation and execution.

6 Conclusion

In this study, we develop EHRAgent, an LLM agent with external tools for few-shot multi-tabular reasoning on real-world EHRs. Empowered by the

emergent few-shot learning capabilities of LLMs, EHRAgent leverages autonomous code generation and execution for direct communication between clinicians and EHR systems. We also improve EHRAgent by interactive coding with execution feedback, along with accumulative medical knowledge, thereby effectively facilitating plan optimization for multi-step problem-solving. Our experiments demonstrate the advantages of EHRAgent over baseline LLM agents in autonomous coding and improved medical reasoning.

Limitation and Future Work

EHRAgent holds considerable potential for positive social impact in a wide range of clinical tasks and applications, including but not limited to patient cohort definition, clinical trial recruitment, case review selection, and treatment decision-making support. Despite the significant improvement in model performance, we have identified several potential limitations of EHRAgent as follows:

Additional Execution Calls. We acknowledge that when compared to open-loop systems such as CoT, Self-Consistency, Chameleon, and LLM2SQL, which generate a complete problem-solving plan at the beginning without any adaptation during execution; EHRAgent, as well as other baselines that rely on environmental feedback like ReAct, Reflexion, Self-Debugging, and AutoGen, require additional LLM calls due to the multi-round conversation. However, such open-loop systems all overlook environmental feedback and cannot adaptively refine their planning processes. These shortcomings largely hinder their performance for the challenging EHR question-answering task, as the success rates for these methods on all three EHR datasets are all below 40%. We can clearly observe the trade-off between performance and execution times. Although environmental feedback enhances performance, future work will focus on cost-effective improvements to balance performance and cost (Zhang et al., 2023).

Translational Clinical Research Considerations. Given the demands for privacy, safety, and ethical considerations in real-world clinical research and practice settings, our goal is to further advance EHRAgent by mitigating biases and addressing ethical implications, thereby contributing to the development of responsible artificial intelligence for healthcare and medicine. Furthermore,

the adaptation and generalization of EHRAgent in low-resource languages is constrained by the availability of relevant resources and training data. Due to limited access to LLMs' API services and constraints related to budget and computation resources, our current experiments are restricted to utilizing the Microsoft Azure OpenAI API service with the gpt-3.5-turbo (0613) and gpt-4 (0613) models. As part of our important future directions, we plan to enhance EHRAgent by incorporating fine-tuned white-box LLMs, such as LLaMA-2 (Touvron et al., 2023).

Completion Rate under Clinical Scenarios. Besides success rate (SR) as our main evaluation metric, we follow Xu et al. (2023); Kirk et al. (2024) and employ completion rate (CR) to denote the percentage of queries for which the model can generate executable plans, irrespective of whether the results are accurate. However, it is important to note that a higher CR may not necessarily imply a superior outcome, especially in clinical settings. In such cases, it is generally preferable to acknowledge failure rather than generate an incorrect answer, as this could lead to an inaccurate diagnosis. We will explore stricter evaluation metrics to assess the cases of misinformation that could pose a risk within clinical settings in our future work.

Privacy and Ethical Statement

In compliance with the PhysioNet Credentialed Health Data Use Agreement 1.5.0[4], we strictly prohibit the transfer of confidential patient data (MIMIC-III and eICU) to third parties, including through online services like APIs. To ensure responsible usage of Azure OpenAI Service based on the guideline[5], we have opted out of the human review process by requesting the Azure OpenAI Additional Use Case Form[6], which prevents third-parties (*e.g.*, Microsoft) from accessing and processing sensitive patient information for any purpose. We continuously and carefully monitor our compliance with these guidelines and the relevant privacy laws to uphold the ethical use of data in our research and operations.

---

[4] https://physionet.org/about/licenses/physionet-credentialed-health-data-license-150/
[5] https://physionet.org/news/post/gpt-responsible-use
[6] https://aka.ms/oai/additionalusecase

Acknowledgments

References

Mahyar Abbasian, Iman Azimi, Amir M. Rahmani, and Ramesh Jain. 2023. Conversational health agents: A personalized llm-powered agent framework.

Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. 2023. Palm 2 technical report.

Seongsu Bae, Daeun Kyung, Jaehee Ryu, Eunbyeol Cho, Gyubok Lee, Sunjun Kweon, Jungwoo Oh, Lei Ji, Eric I-Chao Chang, Tackeun Kim, and Edward Choi. 2023. EHRXQA: A multi-modal question answering dataset for electronic health records with chest x-ray images. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Duane Bender and Kamran Sartipi. 2013. Hl7 fhir: An agile and restful approach to healthcare information exchange. In *Proceedings of the 26th IEEE international symposium on computer-based medical systems*, pages 326–331. IEEE.

Daniil A Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. 2023. Autonomous chemical research with large language models. *Nature*, 624(7992):570–578.

Andres M Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew White, and Philippe Schwaller. 2023. Augmenting large language models with chemistry tools. In *NeurIPS 2023 AI for Science Workshop*.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.

Jerry Cheung, Yuchen Zhuang, Yinghao Li, Pranav Shetty, Wantian Zhao, Sanjeev Grampurohit, Rampi Ramprasad, and Chao Zhang. 2024. POLYIE: A dataset of information extraction from polymer material scientific literature. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 2370–2385, Mexico City, Mexico. Association for Computational Linguistics.

Martin R Cowie, Juuso I Blomster, Lesley H Curtis, Sylvie Duclaux, Ian Ford, Fleur Fritz, Samantha Goldman, Salim Janmohamed, Jörg Kreuzer, Mark Leenay, et al. 2017. Electronic health records to facilitate clinical research. *Clinical Research in Cardiology*, 106:1–9.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Shanghua Gao, Ada Fang, Yepeng Huang, Valentina Giunchiglia, Ayush Noori, Jonathan Richard Schwarz, Yasha Ektefaie, Jovana Kondic, and Marinka Zitnik. 2024. Empowering biomedical discovery with ai agents.

Ran Gong, Qiuyuan Huang, Xiaojian Ma, Hoi Vo, Zane Durante, Yusuke Noda, Zilong Zheng, Song-Chun Zhu, Demetri Terzopoulos, Li Fei-Fei, et al. 2023. Mindagent: Emergent gaming interaction. *ArXiv preprint*, abs/2309.09971.

Tracy D Gunter and Nicolas P Terry. 2005. The emergence of national electronic health record architectures in the united states and australia: models, costs, and questions. *Journal of medical Internet research*, 7(1):e383.

Lavender Yao Jiang, Xujin Chris Liu, Nima Pour Nejatian, Mustafa Nasir-Moin, Duo Wang, Anas Abidin, Kevin Eaton, Howard Antony Riina, Ilya Laufer, Paawan Punjabi, et al. 2023. Health system-scale language models are all-purpose prediction engines. *Nature*, pages 1–6.

Qiao Jin, Yifan Yang, Qingyu Chen, and Zhiyong Lu. 2024. GeneGPT: augmenting large language models with domain tools for improved access to biomedical information. *Bioinformatics*, 40(2):btae075.

Alistair EW Johnson, Tom J Pollard, Lu Shen, Li-wei H Lehman, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G Mark. 2016. Mimic-iii, a freely accessible critical care database. *Scientific data*, 3(1):1–9.

James R Kirk, Robert E Wray, Peter Lindes, and John E Laird. 2024. Improving knowledge extraction from llms for task learning through agent analysis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 18390–18398.

Gyubok Lee, Hyeonji Hwang, Seongsu Bae, Yeonsu Kwon, Woncheol Shin, Seongjun Yang, Minjoon Seo, Jong-Yeup Kim, and Edward Choi. 2022. EHRSQL: A practical text-to-SQL benchmark for electronic health records. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023a. CAMEL: Communicative agents for "mind" exploration of large language model society. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023b. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3102–3116, Singapore. Association for Computational Linguistics.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500. IEEE.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-play compositional reasoning with large language models. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Joshua C Mandel, David A Kreda, Kenneth D Mandl, Isaac S Kohane, and Rachel B Ramoni. 2016. Smart on fhir: a standards-based, interoperable apps platform for electronic health records. *Journal of the American Medical Informatics Association*, 23(5):899–908.

Michael Moor, Oishi Banerjee, Zahra Shakeri Hossein Abad, Harlan M Krumholz, Jure Leskovec, Eric J Topol, and Pranav Rajpurkar. 2023. Foundation models for generalist medical artificial intelligence. *Nature*, 616(7956):259–265.

Linyong Nan, Ellen Zhang, Weijin Zou, Yilun Zhao, Wenfei Zhou, and Arman Cohan. 2023. On evaluating the integration of reasoning and action in llm agents with database question answering.

OpenAI. 2023. Gpt-4 technical report. *arXiv*.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive apis.

Tom J. Pollard, Alistair E. W. Johnson, Jesse D. Raffa, Leo A. Celi, Roger G. Mark, and Omar Badawi. 2018. The eICU collaborative research database, a freely available multi-center database for critical care research. *Scientific Data*, 5(1):180178.

Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Shuofei Qiao, Honghao Gui, Chengfei Lv, Qianghuai Jia, Huajun Chen, and Ningyu Zhang. 2023. Making language models better tool learners with execution feedback. *ArXiv preprint*, abs/2305.13068.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, et al. 2023. Tool learning with foundation models.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*.

Preethi Raghavan, Jennifer J Liang, Diwakar Mahajan, Rachita Chandra, and Peter Szolovits. 2021. emrkbqa: A clinical knowledge-base question answering dataset. In *Proceedings of the 20th Workshop on Biomedical Language Processing*, pages 64–73.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugging-GPT: Solving AI tasks with chatGPT and its friends in hugging face. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. 2023. Adaplanner: Adaptive planning from feedback with language models. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Xiangru Tang, Anni Zou, Zhuosheng Zhang, Yilun Zhao, Xingyao Zhang, Arman Cohan, and Mark Gerstein. 2023. Medagents: Large language models as collaborators for zero-shot medical reasoning.

Surendrabikram Thapa and Surabhi Adhikari. 2023. Chatgpt, bard, and large language models for biomedical research: opportunities and pitfalls. *Annals of Biomedical Engineering*, 51(12):2647–2651.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. 2023a. A survey on large language model based autonomous agents.

Ping Wang, Tian Shi, and Chandan K Reddy. 2020. Text-to-sql generation for question answering on electronic medical records. In *Proceedings of The Web Conference 2020*, pages 350–361.

Weizhi Wang, Li Dong, Hao Cheng, Xiaodong Liu, Xifeng Yan, Jianfeng Gao, and Furu Wei. 2023b. Augmenting language models with long-term memory. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Xiaoxuan Wang, Ziniu Hu, Pan Lu, Yanqiao Zhu, Jieyu Zhang, Satyen Subramaniam, Arjun R. Loomba, Shichang Zhang, Yizhou Sun, and Wei Wang. 2023c. Scibench: Evaluating college-level scientific problem-solving abilities of large language models.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024a. Executable code actions elicit better LLM agents. In *Forty-first International Conference on Machine Learning*.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2024b. MINT: Evaluating LLMs in multi-turn interaction with tools and language feedback. In *The Twelfth International Conference on Learning Representations*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023d. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. 2023. On the tool manipulation capability of open-source large language models.

John Yang, Akshara Prabhakar, Karthik R Narasimhan, and Shunyu Yao. 2023. Intercode: Standardizing and benchmarking interactive coding with execution feedback. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Xi Yang, Aokun Chen, Nima PourNejatian, Hoo Chang Shin, Kaleb E Smith, Christopher Parisien, Colin Compas, Cheryl Martin, Anthony B Costa, Mona G Flores, et al. 2022. A large language model for electronic health records. *NPJ Digital Medicine*, 5(1):194.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik R Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023b. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R Fung, Hao Peng, and Heng Ji. 2024a. CRAFT: Customizing LLMs by creating and retrieving from specialized toolsets. In *The Twelfth International Conference on Learning Representations*.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Ren Kan, Dongsheng Li, and Deqing Yang. 2024b. Easytool: Enhancing llm-based agents with concise tool instruction.

Jieyu Zhang, Ranjay Krishna, Ahmed H. Awadallah, and Chi Wang. 2023. Ecoassistant: Using llm assistant more affordably and accurately.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2023. Expel: Llm agents are experiential learners.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning.

Yuchen Zhuang, Xiang Chen, Tong Yu, Saayan Mitra, Victor Bursztyn, Ryan A. Rossi, Somdeb Sarkhel, and Chao Zhang. 2024. Toolchain*: Efficient action space navigation in large language models with a* search. In *The Twelfth International Conference on Learning Representations*.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. Toolqa: A dataset for llm question answering with external tools. *Advances in Neural Information Processing Systems*, 36:50117–50143.

A Dataset and Task Details

A.1 Task Details

We evaluate `EHRAgent` on three publicly available EHR datasets from two text-to-SQL medical question answering (QA) benchmarks (Lee et al., 2022), EHRSQL[7] and TREQS[8], built upon structured EHRs from MIMIC-III and eICU. EHRSQL and TREQS serve as text-to-SQL benchmarks for assessing the performance of medical QA models, specifically focusing on generating SQL queries for addressing a wide range of real-world questions gathered from over 200 hospital staff. Questions within EHRSQL and TREQS, ranging from simple data retrieval to complex operations such as calculations, reflect the *diverse* and *complex* clinical tasks encountered by front-line healthcare professionals. Dataset statistics are available in Table 3.

Dataset	# Examples	# Table	# Row/Table	# Table/Q
MIMIC-III	580	17	81k	2.52
eICU	580	10	152k	1.74
TREQS	996	5	498k	1.48
Average	**718.7**	**10.7**	**243.7k**	**1.91**

Table 3: Dataset statistics.

A.2 Question Complexity Level

We categorize input queries into various complexity levels (levels I-IV for MIMIC-III and levels I-III for eICU and TREQS) based on the number of tables involved in solution generation. For example, given the question 'How many patients were given temporary tracheostomy?', the complexity level is categorized as II, indicating that we need to extract information from two tables (admission and procedure) to generate the solution. Furthermore, we also conduct a performance analysis (see Figure 4) based on additional evaluation metrics related to question complexity, including (1) the number of elements (*i.e.*, slots) in each question and (2) the number of columns involved in each solution. Specifically, elements refer to the slots within each template that can be populated with pre-defined values or database records.

A.3 MIMIC-III

MIMIC-III (Johnson et al., 2016)[9] covers 38,597 patients and 49,785 hospital admissions information in critical care units at the Beth Israel Deaconess Medical Center ranging from 2001 to 2012. It includes deidentified administrative information such as demographics and highly granular clinical information, including vital signs, laboratory results, procedures, medications, caregiver notes, imaging reports, and mortality.

A.4 eICU

Similar to MIMIC-III, eICU (Pollard et al., 2018)[10] includes over 200,000 admissions from multiple critical care units across the United States in 2014 and 2015. It contains deidentified administrative information following the US Health Insurance Portability and Accountability Act (HIPAA) standard and structured clinical data, including vital signs, laboratory measurements, medications, treatment plans, admission diagnoses, and medical histories.

A.5 TREQS

TREQS (Wang et al., 2020) is a healthcare question and answering benchmark that is built upon the MIMIC-III (Johnson et al., 2016) dataset. In TREQS, questions are generated automatically using pre-defined templates with the text-to-SQL task. Compared to the MIMIC-III dataset within the EHRSQL (Lee et al., 2022) benchmark, TREQS has a narrower focus in terms of the types of questions and the complexity of SQL queries. Specifically, it is restricted to only five tables but includes

---

[7]https://github.com/glee4810/EHRSQL
[8]https://github.com/wangpinggl/TREQS

[9]https://physionet.org/content/mimiciii/1.4/
[10]https://physionet.org/content/eicu-crd/2.0/

a significantly larger number of records (Table 3) within each table.

B   Tool Set Details

To obtain relevant information from EHRs and enhance the problem-solving capabilities of LLM-based agents, we augment LLMs with the following tools:

◇ **Database Loader** loads a specific table from the database.

◇ **Data Filter** applies specific filtering condition to the selected table. These conditions are defined by a column name and a relational operator. The relational operator may take the form of a comparison (e.g., "<" or ">") with a specific value, either with the column's values or the count of values grouped by another column. Alternatively, it could be operations such as identifying the minimum or maximum values within the column.

◇ **Get Value** retrieves either all the values within a specific column or performs basic operations on all the values, including calculations for the mean, maximum, minimum, sum, and count.

◇ **Calculator** calculates the results from input strings. We leverage the WolframAlpha API portal[11], which can handle both straightforward calculations such as addition, subtraction, and multiplication and more complex operations like averaging and identifying maximum values.

◇ **Date Calculator** calculates the target date based on the input date and the provided time interval information.

◇ **SQL Interpreter** interprets and executes SQL code written by LLMs.

C   Baseline Details

All the methods, including baselines and EHRAgent, share the same *(1) tool definitions, (2) table meta information, and (3) few-shot demonstrations* in the prompts by default. The only difference is the prompting style or technical differences between different methods, which guarantees a *fair comparison* among all baselines and EHRAgent. Table 4 summarizes the inclusion of different components in both baselines and ours.

• **Baselines w/o Code Interface.** LLMs without a code interface rely purely on natural language-based planning capabilities.

◇ **CoT** (Wei et al., 2022): CoT enhances the complex reasoning capabilities of original LLMs by

---

generating a series of intermediate reasoning steps.

◇ **Self-Consistency** (Wang et al., 2023d): Self-consistency improves CoT by sampling diverse reasoning paths to replace the native greedy decoding and select the most consistent answer.

◇ **Chameleon** (Lu et al., 2023): Chameleon employs LLMs as controllers and integrates a set of plug-and-play modules, enabling enhanced reasoning and problem-solving across diverse tasks.

◇ **ReAct** (Yao et al., 2023b): ReAct integrates reasoning with tool use by guiding LLMs to generate intermediate verbal reasoning traces and tool commands.

◇ **Reflexion** (Shinn et al., 2023): Reflexion leverages verbal reinforcement to teach LLM-based agents to learn from linguistic feedback from past mistakes.

• **Baselines w/ Code Interface.** LLMs with a code interface enhance the inherent capabilities of LLMs by enabling their interaction with programming languages and the execution of code. In accordance with their default configuration, we present a summary of the utilization of programming languages in various baselines in Table 5. Additionally, we provide a detailed explanation of the programming language selection in EHRAgent in Appendix D.

◇ **LLM2SQL** (Nan et al., 2023): LLM2SQL augments LLMs with a code interface to generate SQL queries for retrieving information from EHRs for question answering.

◇ **DIN-SQL** (Pourreza and Rafiei, 2023): Compared to LLM2SQL, DIN-SQL further breaks down a complex problem into several sub-problems and feeding the solutions of those sub-problems into LLMs, effectively improving problem-solving performance.

◇ **Self-Debugging** (Chen et al., 2024): Self-Debugging teaches LLMs to debug by investigating execution results and explaining the generated code in natural language.

◇ **AutoGen** (Wu et al., 2023): AutoGen unifies LLM-based agent workflows as multi-agent conversations and uses the code interface to encode interactions between agents and environments. We follow the official tutorial[12] for the implementation of AutoGen. Specifically, we utilize the built-in AssistantAgent and UserProxyAgent within AutoGen to serve as the LLM agent and the code executor, respectively. The AssistantAgent is

---

[11]https://products.wolframalpha.com/api

[12]https://microsoft.github.io/autogen/docs/Use-Cases/agent_chat/

Baselines	Tool Use	Code Interface	Environment Feedback	Debugging	Error Exploration	Medical Information	Long-term Memory
w/o Code Interface							
CoT (Wei et al., 2022)	✓	✗	✗	✗	✗	✗	✗
Self-Consistency (Wang et al., 2023d)	✓	✗	✗	✗	✗	✗	✗
Chameleon (Lu et al., 2023)	✓	✗	✗	✗	✗	✗	✗
ReAct (Yao et al., 2023b)	✓	✗	✓	✗	✗	✗	✗
Reflexion (Shinn et al., 2023)	✓	✗	✓	✓	✗	✗	✗
w/ Code Interface							
LLM2SQL (Nan et al., 2023)	✗	✓	✗	✗	✗	✗	✗
DIN-SQL (Pourreza and Rafiei, 2023)	✗	✓	✗	✗	✗	✗	✗
Self-Debugging (Chen et al., 2024)	✗	✓	✓	✓	✗	✗	✗
AutoGen (Wu et al., 2023)	✓	✓	✓	✓	✗	✗	✗
EHRAgent (Ours)	✓	✓	✓	✓	✓	✓	✓

Table 4: Comparison of baselines and EHRAgent on the inclusion of different components.

configured in accordance with AutoGen's tailored system prompts, which are designed to direct the LLM to (1) propose code within a coding block as required, (2) refine the proposed code based on execution outcomes, and (3) append a specific code, "TERMINATE", to conclude the response for terminating the dialogue. The UserProxyAgent functions as a surrogate for the user, extracting and executing code from the LLM's responses in a local environment. Subsequently, it relays the execution results back to the LLM. In instances where code is not detected, a standard message is dispatched instead. This arrangement facilitates an automated dialogue process, obviating the need for manual tasks such as code copying, pasting, and execution by the user, who only needs to initiate the conversation with an original query.

Baselines	# Language
LLM2SQL (Nan et al., 2023)	SQL
DIN-SQL (Pourreza and Rafiei, 2023)	SQL
Self-Debugging (Chen et al., 2024)	SQL
AutoGen (Wu et al., 2023)	Python
EHRAgent (Ours)	**Python**

Table 5: Comparison of baselines and EHRAgent on the selection of primary programming languages.

D Selection of Primary Programming Language

In our main experiments, we concentrate on three SQL-based EHR QA datasets to assess EHRAgent in comparison with other baselines. Nevertheless, we have opted for Python as the primary programming language for EHRAgent, rather than SQL[13].

---

[13] We include an empirical analysis in Appendix G.3 to further justify the selection of Python as the primary program-

The primary reasons for *choosing Python instead of SQL* to address medical inquiries based on EHRs are outlined below:

Python Enables the External Tool-Use. Using alternative programming languages, such as SQL, can result in LLM-based agents becoming unavailable to external tools or functions. The primary contribution of EHRAgent is to develop a code-empowered agent capable of generating and executing code-based plans to solve complex real-world clinic tasks. In general, the SQL language itself is incapable of calling API functions. For example, EHRXQA (Bae et al., 2023) can be considered as an LLM agent that generates a solution plan in NeuralSQL (not SQL). This agent is equipped with two tools: a pre-trained Visual Question Answering (VQA) model called FUNC_VQA, and a SQL interpreter. Similar to EHRAgent, it also relies on a non-SQL language and includes an SQL interpreter as a tool. Compared with NeuralSQL in EHRXQA (Bae et al., 2023), Python in EHRAgent can be directly executed, while NeuralSQL requires additional parsing.

Python Enables the Integration of SQL Tool Function. Python provides excellent interoperability with various databases and data formats. It supports a wide range of database connectors, including popular relational databases such as PostgreSQL, MySQL, and SQLite, as well as non-relational databases like MongoDB. This interoperability ensures that EHRAgent can seamlessly interact with different EHR systems and databases. Although our proposed method primarily relies on generating and executing

---

ming language for EHRAgent.

Python code, *we do not prohibit* EHRAgent *from utilizing SQL to solve problems.* In our prompts and instructions, we also provide the 'SQLInterpreter' tool function for the agent to perform relational database operations using SQL. Through our experiments, we have observed that EHRAgent is capable of combining results from Python code and SQL commands effectively. For instance, when presented with the question, âĂIJShow me patient 28020's length of stay of the last hospital stay.âĂİ, EHRAgent will first generate SQL command `admit_disch_tuple = SQLInterpreter(ŚELECT ADMITTIME, DISCHTIME FROM admissions WHERE SUBJECT_ID=28020 ORDER BY ADMITTIME DESC LIMIT 1́)` and execute it to obtain the tuples containing the patient's admission and discharge times. It will then employ Python code along with the built-in date-time function to calculate the duration of the last stay tuple.

Python Enables a More Generalizable Framework. EHRAgent is a generalizable LLM agent empowered with a code interface to autonomously generate and execute code as solutions to given problems. While Section 4 focuses on the challenging multi-tabular reasoning task within EHRs for evaluation, the Python-based approach has the potential to be generalized to other tasks (e.g., risk prediction tasks based on EHRs) or even multi-modal clinical data and be integrated with additional toolsets in the future. In contrast, other languages like SQL are limited to database-related operations.

Python is More Flexible in Extension. Python is a general-purpose programming language that offers greater flexibility compared to SQL. It enables the implementation of complex logic and algorithms, which may be necessary for solving certain types of medical questions that require more than simple database queries. Python is also a highly flexible programming language that offers extensive capabilities through its libraries and frameworks, making it suitable for handling a wide range of programming tasks, including database operations. In contrast, SQL is only applicable within relational databases and does not provide the same level of flexibility and extension. This attribute is particularly important to LLM-based agents, as they can leverage both existing Python libraries and custom-defined functions as tools to solve complex problems that are inaccessible for and beyond the scope of SQL.

Python Includes More Extensive Resources for Pre-Training. Python has a large and active community of developers and researchers. This community contributes to the development of powerful libraries, frameworks, and tools that can be leveraged in EHRAgent. The extensive documentation, tutorials, and forums available for Python also provide valuable resources for troubleshooting and optimization. **Github repositories are one of the most extensive sources of code data for state-of-the-art language models (*i.e.*, LLMs), such as GPTs.** Python is the most widely used coding language on Github[14]. In addition, Python is known for its readability and maintainability. The clean and expressive syntax of Python makes it easier for researchers and developers to understand, modify, and extend the codebase of EHRAgent. This is particularly important when extended to real-world clinical research and practice, where the system may need to be updated frequently to incorporate new knowledge and adapt to evolving requirements.

E Additional Implementation Details

E.1 Hardware and Software Details

All experiments are conducted on CPU: Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz and GPU: NVIDIA GeForce RTX A5000 GPUs, using Python 3.9 and AutoGen 0.2.0[15].

E.2 Data Preprocessing Details

During the data pre-processing stage, we create EHR question-answering pairs by considering text queries as questions and executing SQL commands in the database to automatically generate the corresponding ground-truth answers. We filter out samples containing unexecutable SQL commands or yielding empty results throughout this process.

E.3 Code Generation Details

Given that the majority of LLMs have been pre-trained on Python code snippets (Gao et al., 2023), and Python's inherent modularity aligns well with tool functions, we choose Python 3.9 as the primary coding language for interaction coding and AutoGen 0.2.0 (Wu et al., 2023) as the interface for communication between the LLM agent and the code executor.

---

[14]https://madnight.github.io/githut/#/pull_requests/2023/1

[15]https://github.com/microsoft/autogen

E.4 Selection of Initial Set of Demonstrations

The initial set of examples is collected manually, following four criteria: (1) using the same demonstrations across all the baselines; (2) utilizing all the designed tools; (3) covering as many distinct tables as possible; and (4) including examples in different styles of questions. With these criteria in mind, we manually crafted four demonstrations for each dataset. To ensure a fair comparison, we use the same initial four-shot demonstrations ($K = 4$) for all baselines and EHRAgent, considering the maximum length limitations of input context in baselines like ReAct (Yao et al., 2023b) and Chameleon (Lu et al., 2023).

E.5 Evaluation Metric Details

Our main evaluation metric is the success rate (SR), quantifying the percentage of queries that the model successfully handles. In addition, we leverage completion rate (CR) as a side evaluation metric to represent the percentage of queries for which the model is able to generate executable plans, regardless of whether the results are correct. Specifically, following existing LLM-based agent studies (Xu et al., 2023; Kirk et al., 2024), we use CR to assess the effectiveness of LLM-based agents in generating complete executable plans without execution errors. One of our key components in EHRAgent is interactive coding with environmental feedback. By using CR, we can demonstrate that our proposed EHRAgent, along with other baselines that incorporate environmental feedback (e.g., ReAct (Yao et al., 2023b), Reflexion (Shinn et al., 2023), Self-Debugging (Chen et al., 2024), and AutoGen (Wu et al., 2023)), has a stronger capability (higher CR) in generating complete executable plans without execution errors, compared to baselines without environmental feedback (e.g., CoT (Wei et al., 2022), Self-Consistency (Wang et al., 2023d), Chameleon (Lu et al., 2023), and LLM2SQL (Nan et al., 2023)).

E.6 EHR Metadata Details

◇ **MIMIC-III.**

```
┌─────── <MIMIC_III> Metadata ───────┐
Read the following data descriptions, generate
    the background knowledge as the context
    information that could be helpful for
    answering the question.
(1) Tables are linked by identifiers which
    usually have the suffix 'ID'. For example,
    SUBJECT_ID refers to a unique patient,
    HADM_ID refers to a unique admission to
```

```
    the hospital, and ICUSTAY_ID refers to a
    unique admission to an intensive care unit.
(2) Charted events such as notes, laboratory
    tests, and fluid balance are stored in a
    series of 'events' tables. For example the
    outputevents table contains all measurements
    related to output for a given patient,
    while the labevents table contains
    laboratory test results for a patient.
(3) Tables prefixed with 'd_' are dictionary
    tables and provide definitions for
    identifiers.c For example, every row of
    chartevents is associated with a single
    ITEMID which represents the concept
    measured, but it does not contain the
    actual name of the measurement. By joining
    chartevents and d_items on ITEMID, it is
    possible to identify the concept represented
    by a given ITEMID.
(4) For the databases, four of them are used to
    define and track patient stays: admissions,
    patients,icustays, and transfers. Another
    four tables are dictionaries for cross-
    referencing codes against their respective
    definitions: d_icd_diagnoses,
    d_icd_procedures, d_items, and d_labitems.
    The remaining tables, including chartevents,
    cost, inputevents_cv, labevents,
    microbiologyevents, outputevents,
    prescriptions, procedures_icd, contain data
    associated with patient care, such as
    physiological measurements, caregiver
    observations, and billing information.
```

◇ **eICU.**

```
┌─────── <eICU> Metadata ───────┐
Read the following data descriptions, generate
    the background knowledge as the context
    information that could be helpful for
    answering the question.
(1) Data include vital signs, laboratory
    measurements, medications, APACHE components,
    care plan information, admission diagnosis,
    patient history, time-stamped diagnoses
    from a structured problem list, and
    similarly chosen treatments.
(2) Data from each patient is collected into a
    common warehouse only if certain interfaces
    are available. Each interface is used to
    transform and load a certain type of data:
    vital sign interfaces incorporate vital
    signs, laboratory interfaces provide
    measurements on blood samples, and so on.
(3) It is important to be aware that different
    care units may have different interfaces in
    place, and that the lack of an interface
    will result in no data being available for
    a given patient, even if those measurements
    were made in reality. The data is provided
    as a relational database, comprising
    multiple tables joined by keys.
(4) All the databases are used to record
    information associated to patient care,
    such as allergy, cost, diagnosis,
    intakeoutput, lab, medication, microlab,
    patient, treatment, vitalperiodic.
```

⋄ **TREQS.**

```
┌──────── <TREQS> Metadata ─────────┐
Read the following data descriptions, generate
    the background knowledge as the context
    information that could be helpful for
    answering the question.
(1) The database contains five categories of
    information for patients, including
    demographics, laboratory tests, diagnosis,
    procedures and prescriptions, and prepared
    a specific table for each category
    separately.
(2) These tables compose a relational patient
    database where tables are linked through
    patient ID and admission ID.
└────────────────────────────────────┘
```

E.7 Prompt Details

In the subsequent subsections, we detail the prompt templates employed in EHRAgent. The complete version of the prompts is available at our code repository due to space limitations.

⋄ **Prompt for Code Generation.** We first present the prompt template for EHRAgent in code generation as follows:

```
┌──────── <LLM_Agent> Prompt ─────────┐
Assume you have knowledge of several tables:
{OVERALL_EHR_DESCRIPTIONS}
Write a python code to solve the given question.
    You can use the following functions:
{TOOL_DEFINITIONS}
Use the variable 'answer' to store the answer
    of the code. Here are some examples:
{4-SHOT_EXAMPLES}
(END OF EXAMPLES)
Knowledge:
{KNOWLEDGE}
Question: {QUESTION}
Solution:
└────────────────────────────────────┘
```

⋄ **Prompt for Knowledge Integration.** We then present the prompt template for knowledge integration in EHRAgent as follows:

```
┌────── <Medical_Knowledge> Prompt ──────┐
Read the following data descriptions, generate
    the background knowledge as the context
    information that could be helpful for
    answering the question.
{OVERALL_EHR_DESCRIPTIONS}
For different tables, they contain the
    following information:
{COLUMNAR_DESCRIPTIONS}

{4-SHOT_EXAMPLES}

Question: {QUESTION}
Knowledge:
└────────────────────────────────────┘
```

⋄ **Prompt for 'Rubber Duck' Debugging.** The prompt template used for debugging module in

EHRAgent is shown as follows:

```
┌────── <Error_Exploration> Prompt ──────┐
Given a question:
{QUESTION}
The user has written code with the following
    functions:
{TOOL_DEFINITIONS}

The code is as follows:
{CODE}

The execution result is:
{ERROR_INFO}

Please check the code and point out the most
    possible reason to the error.
└────────────────────────────────────┘
```

⋄ **Prompt for Few-Shot Examples.** The prompt template used for few-shot examples in EHRAgent is shown as follows:

```
┌────── <Few_Shot_Examples> Prompt ──────┐
Question: {QUESTION_I}
Knowledge:
{KNOWLEDGE_I}
Solution: {CODE_I}

Question: {QUESTION_II}
Knowledge:
{KNOWLEDGE_II}
Solution: {CODE_II}

Question: {QUESTION_III}
Knowledge:
{KNOWLEDGE_III}
Solution: {CODE_III}

Question: {QUESTION_IV}
Knowledge:
{KNOWLEDGE_IV}
Solution: {CODE_IV}
└────────────────────────────────────┘
```

F Additional Experimental Results

F.1 Effect of Base LLMs

Table 6 presents a summary of the experimental results obtained from EHRAgent and all baselines using a different base LLM, GPT-3.5-turbo (0613). The results clearly demonstrate that EHRAgent continues to outperform all the baselines, achieving a performance gain of 6.72%. This highlights the ability of EHRAgent to generalize across different base LLMs as backbone models. When comparing the experiments conducted with GPT-4 (Table 1), the performance of both the baselines and EHRAgent decreases. This can primarily be attributed to the weaker capabilities of instruction-following and reasoning in GPT-3.5-turbo.

Dataset (→)	MIMIC-III					
Complexity Level (→)	I	II	III	IV	All	
Methods (↓) /Metrics (→)			SR.		SR.	CR.
w/o Code Interface						
CoT (Wei et al., 2022)	23.16	10.40	2.99	1.71	8.62	41.55
Self-Consistency (Wang et al., 2023d)	25.26	11.88	4.19	2.56	10.52	47.59
Chameleon (Lu et al., 2023)	27.37	11.88	3.59	2.56	11.21	47.59
ReAct (Yao et al., 2023b)	26.32	10.89	3.59	3.42	9.66	61.21
Reflexion (Shinn et al., 2023)	30.53	12.38	9.58	8.55	13.28	66.72
w/ Code Interface						
LLM2SQL (Nan et al., 2023)	21.05	15.84	4.19	2.56	10.69	59.49
Self-Debugging (Chen et al., 2024)	36.84	33.66	22.75	16.24	27.59	72.93
AutoGen (Wu et al., 2023)	28.42	25.74	13.17	10.26	19.48	52.42
EHRAgent (Ours)	**43.16**	**42.57**	**29.94**	**18.80**	**34.31**	**78.80**

Table 6: Experimental results of success rate (*i.e.*, SR.) and completion rate (*i.e.*, CR.) on MIMIC-III using `GPT-3.5-turbo` as the base LLM. The complexity of questions increases from Level I (the simplest) to Level IV (the most difficult).

F.2 Additional Ablation Studies

We conduct additional ablation studies to evaluate the effectiveness of each module in EHRAgent on eICU in Table 7 and obtain consistent results. From the results from both MIMIC-III and eICU, we observe that all four components contribute significantly to the performance gain.

◇ **Medical Information Integration.** Out of all the components, the medical knowledge injection module mainly exhibits its benefits in challenging tasks. These tasks often involve more tables and require a deeper understanding of domain knowledge to associate items with their corresponding tables.

◇ **Long-term Memory.** Following the reinforcement learning setting (Sun et al., 2023; Shinn et al., 2023), the long-term memory mechanism improves performance by justifying the necessity of selecting the most relevant demonstrations for planning. In order to simulate the scenario where the ground truth annotations (*i.e.*, rewards) are unavailable, we further evaluate the effectiveness of the long-term memory on the completed cases in Table 8, regardless of whether they are successful or not. The results indicate that the inclusion of long-term memory with completed cases increases the completion rate but tends to reduce the success rate across most difficulty levels, as some incorrect cases might be included as the few-shot demonstrations. We have also performed multi-round experiments with shuffled order and observed that the order had almost no influence on the final performance in all three datasets. Nonetheless, it still outperforms the per-

formance without long-term memory, confirming the effectiveness of the memory mechanism.

◇ **Interactive Coding.** For the ablation study setting of EHRAgent w/o interactive coding, we directly chose CoT (Wei et al., 2022) as the backbone, where we deteriorate from generating code-based plans to *natural language-based plans*. Once the steps are generated, we execute them in a step-by-step manner and *obtain error information from the tool functions*. By combining the error messages with tool definitions and language-based plans, we are still able to prompt the LLMs to deduce the most probable underlying cause of the error. The medical information injection and long-term memory components remain unchanged from the original EHRAgent. From the ablation studies, we can observe that the interactive coding interface is the most significant contributor to the performance gain across all complexity levels. This verifies the importance of utilizing the code interface for planning instead of natural languages, which enables the model to avoid overly complex contexts and thus leads to a substantial increase in the completion rate. Additionally, the code interface also allows the debugging module to refine the planning with execution feedback, improving the efficacy of the planning process.

◇ **Debugging Module.** The 'rubber duck' debugging module enhances the performance by guiding the LLM agent to figure out the underlying reasons for the error messages. This enables EHRAgent to address the intrinsic error that occurs in the original reasoning steps. We then further illustrate the difference between debugging modules in EHRAgent and others. Self-debugging (Chen et al., 2024) that sends back the execution results with an explanation of the code for plan refinement. Reflexion (Shinn et al., 2023) sends the binary reward of whether it is successful or not back for refinement, which contains little information. In both cases, however, the error message is still information on the surface, like âĂŸincorrect queryâĂŹ, etc. This is aligned with our empirical observations that LLM agents tend to make slight modifications to the code snippets based on the error message without further debugging. Taking one step further, our debugging module in EHRAgent incorporates an error tracing procedure that enables the LLM to analyze potential causes beyond the current error message. Our debugging module aims to leverage the conversation format to think one step further about potential reasons, such as âĂŸincorrect col-

umn names in the queryâĂŹ or âĂŸincorrect values in the queryâĂŹ.

Complexity level	I	II	III	All	
Metrics		SR.		SR.	CR.
EHRAgent	**54.82**	**53.52**	**25.00**	**53.10**	**91.72**
w/o medical information	36.75	28.39	6.25	30.17	47.24
w/o long-term memory	52.41	44.22	18.75	45.69	78.97
w/o interactive coding	46.39	44.97	6.25	44.31	65.34
w/o rubber duck debugging	50.60	46.98	12.50	47.07	70.86

Table 7: Additional ablation studies on success rate (*i.e.*, SR.) and completion rate (*i.e.*, CR.) under different question complexity (I-III) on eICU dataset.

Complexity level	I	II	III	IV	All	
Metrics			SR.		SR.	CR.
EHRAgent (LTM w/ Success)	71.58	**66.34**	**49.70**	**49.14**	**58.97**	85.86
LTM w/ Completion	**76.84**	60.89	41.92	34.48	53.24	**90.05**
w/o LTM	65.96	54.46	37.13	42.74	51.73	83.42

Table 8: Comparison on long-term memory (*i.e.*, LTM) design under different question complexity (I-IV) on MIMIC-III dataset.

F.3 Cost Estimation

Using GPT-4 as the foundational LLM model, we report the average cost of EHRAgent for each query in the MIMIC-III, eICU, and TREQS datasets as $0.60, $0.17, and $0.52, respectively. The cost is mainly determined by the complexity of the question (*i.e.*, the number of tables required to answer the question) and the difficulty in locating relevant information within each table.

G Additional Empirical Analysis

G.1 Additional Question Complexity Analysis

We further analyze the model performance by considering various measures of question complexity based on the number of elements in questions, and the number of columns involved in solutions, as shown in Figure 4. Incorporating more elements requires the model to either perform calculations or utilize domain knowledge to establish connections between elements and specific columns. Similarly, involving more columns also presents a challenge for the model in accurately locating and associating the relevant columns. We notice that both EHRAgent and baselines generally exhibit lower performance on more challenging tasks[16]. Notably,

---

[16]Exceptions may exist when considering questions of seven elements in Figures 4(a) and 4(b), as it comprises only

our model consistently outperforms all the baseline models across all levels of difficulty. Specifically, for those questions with more than 10 columns, the completion rate of those open-loop baselines is very low (less than 20%), whereas EHRAgent can still correctly answer around 50% of queries, indicating the robustness of EHRAgent in handling complex queries with multiple elements.

G.2 Additional Error Analysis

We conducted a manual examination to analyze all incorrect cases generated by EHRAgent in MIMIC-III. Figure 6 illustrates the percentage of each type of error frequently encountered during solution generation:

◇ **Date/Time.** When addressing queries related to dates and times, it is important for the LLM agent to use the 'Calendar' tool, which bases its calculations on the system time of the database. This approach is typically reliable, but there are situations where the agent defaults to calculating dates based on real-world time. Such instances may lead to potential inaccuracies.

◇ **Context Length.** This type of error occurs when the input queries or dialog histories are excessively long, exceeding the context length limit.

◇ **Incorrect Logic.** When solving multi-hop reasoning questions across multiple databases, the LLM agent may generate executable plans that contain logical errors in the intermediate reasoning steps. For instance, in computing the total cost of a hospital visit, the LLM agent might erroneously generate a plan that filters the database using patient_id instead of the correct admission_id.

◇ **Incorrect SQL Command.** This error type arises when the LLM agent attempts to integrate the SQLInterpreter into a Python-based plan to derive intermediate results. Typically, incorrect SQL commands result in empty responses from SQLInterpreter, leading to the failure of subsequent parts of the plan.

◇ **Fail to Follow Instructions.** The LLM agent often fails to follow the instructions provided in the initial prompt or during the interactive debugging process.

◇ **Fail to Debug.** Despite undertaking all T-step trials, the LLM agent consistently fails to identify the root cause of errors, resulting in plans that are either incomplete or inexcusable.

---

eight samples and may not be as representative.

G.3 Additional Empirical Comparison of Primary Programming Languages

We conduct an additional analysis based on the empirical results (byond main results in Table 1) to further justify the selection of Python as our primary programming language.

Data Complexity. The SPIDER (Yu et al., 2018) dataset, which is commonly used in SQL baselines (Pourreza and Rafiei, 2023), typically only involves referencing information from *an average of 1.1 tables per question*. In contrast, the EHRQA datasets we utilized require referencing information from *an average of 1.9 tables per question*. This significant gap in # tablesquestions indicates that EHRQA requires more advanced reasoning across multiple tables.

Sample Efficiency. SQL-based methods require more demonstrations. As SQL occupies a relatively smaller proportion of training data, it is quite difficult for LLMs to generate valid SQL commands. Usually, the methods need at least tens of demonstrations to get the LLMs familiar with the data schema and SQL grammar. In EHRAgents, we only need four demonstrations as few-shot multi-tabular reasoning.

Environment Feedback. DIN-SQL (Pourreza and Rafiei, 2023) establishes a set of rules to automatically self-correct the SQL commands generated. Nevertheless, these rules are rigid and may not cover all potential scenarios. While it does contribute to enhancing the validity of the generated SQL commands to some extent, DIN-SQL lacks tailored information to optimize the code based on different circumstances, resulting in a lower success rate compared to self-debugging and EHRAgent, which provide error messages and deeper insights.

Execution Time Efficiency. We acknowledge that when handling large amounts of data, Python may experience efficiency issues compared to SQL commands. We have also observed similar challenges when working with the TREQS dataset, which contains a massive database with millions of records. However, in the MIMIC-III dataset, EHRAgent (avg. 52.63 seconds per question) still demonstrates higher efficiency compared to the state-of-the-art LLM4SQL method, DIN-SQL (Pourreza and Rafiei, 2023) (avg. 103.28 seconds per question). We will consider the efficiency of Python when dealing with large-scale databases

as one of the important future directions.

H Additional Case Studies

We present additional case studies to showcase the effectiveness of each module in EHRAgent, including medical information integration (Section H.1), long-term memory (Section H.2), code interface (Section H.3), and rubber duck debugging module (Section H.4). In addition, we include Figures 13 and 14 to showcase the entire workflow of EHRAgent with accumulative domain knowledge and coding capability for EHR reasoning.

H.1 Case Study of Medical Information Integration

Figure 8 presents a case study of integrated medical knowledge based on a given query. When faced with a question related to *'aspirin ec'* and *'venous cath nec'*, EHRAgent effectively summarizes and integrates relevant knowledge containing drug and procedure information, successfully reasoning and identifying the potential locations of the required information (tables and records) in the EHR database (*e.g.*, *'prescriptions'*, *'d_icd_procedures'*), along with the detailed identifiers (*e.g.*, *'ICD9_CODE'*, *'HADM_ID'*) needed to associate them.

```
Question: count the number of patients who were prescribed
aspirin ec within 2 month after having received a venous
cath nec procedure until 4 year ago.

                        EHRAgent
Knowledge:
- As aspirin ec is a drug, we can find the corresponding
information in the prescriptions database.
- As venous cath nec is a procedure, we can find the
corresponding ICD9_CODE in the d_icd_procedures database.
- The ICD9_CODE can be used to find the corresponding HADM_ID in
the procedures_icd database.
- The HADM_ID can be used to find the corresponding SUBJECT_ID in
the admissions database.
- We can then count the number of unique SUBJECT_IDs in the
prescriptions database who were prescribed aspirin ec within 2
months after having received a venous cath nec procedure until 4
years ago.
```

Figure 8: Case study of medical information injection in EHRAgent on MIMIC-III dataset. Given a question related to *'aspirin ec'* and *'venous cath nec'*, EHRAgent effectively integrates knowledge about their potential location in the database and the identifiers required to associate them.

H.2 Case Study of Long-Term Memory

Figure 9 presents a case study of updating few-shot demonstrations from long-term memory. Due to the constraints of limited context length, we are only able to provide a limited number of examples to guide EHRAgent in generating solution code. For

a given question, the initial set of examples is pre-defined and fixed, which may not cover the specific reasoning logic or knowledge required to solve it. Using long-term memory, EHRAgent replaces original few-shot demonstrations with the most relevant successful cases from past experiences for effective plan refinement. For example, none of the original few-shot examples relate to either 'count the number' scenarios or procedure knowledge; after selecting from the long-term memory pool, we successfully retrieve more relevant examples, thus providing a similar solution logic for reference.

H.3 Case Study of Code Interface

Figures 10 and 11 present two case studies of harnessing LLMs as autonomous agents in a multi-turn conversation for code generation, in comparison to a natural language-based plan such as ReAct. From the case studies, we can observe that ReAct lacks a code interface, which prevents it from utilizing code structures for efficient action planning and tool usage. This limitation often results in a lengthy context for ReAct to execute, which eventually leads to a low completion rate.

H.4 Case Study of Rubber Duck Debugging

Figure 12 showcases a case study comparing the interactive coding process between AutoGen and EHRAgent for the same given query. When executed with error feedback, AutoGen directly sends back the original error messages, making slight modifications (*e.g.*, changing the surface string of the arguments) without reasoning the root cause of the error. In contrast, EHRAgent can identify the underlying causes of the errors through interactive coding and debugging processes. It successfully discovers the underlying error causes (taking into account case sensitivity), facilitating accurate code refinement.

Question: count the number of times that patient 85895 received a ph lab test last month.

Original Examples	Examples from Long-Term Memory
Question: What is the maximum total hospital cost that involves a diagnosis named comp-oth vasc dev/graft since 1 year ago? Knowledge: {KNOWLEDGE} Solution: {SOLUTION}	Question: Count the number of times that patient 52898 were prescribed ns this month. Knowledge: {KNOWLEDGE} Solution: {SOLUTION}
Question: Had any tpn w/lipids been given to patient 2238 in their last hospital visit? Knowledge: {KNOWLEDGE} Solution: {SOLUTION}	Question: Count the number of times that patient 14035 had a d10w intake. Knowledge: {KNOWLEDGE} Solution: {SOLUTION}
Question: What was the name of the procedure that was given two or more times to patient 58730? Knowledge: {KNOWLEDGE} Solution: {SOLUTION}	Question: Count the number of times that patient 99791 received a op red-int fix rad/ulna procedure. Knowledge: {KNOWLEDGE} Solution: {SOLUTION}
Question: What was the last time patient 4718 had a peripheral blood lymphocytes microbiology test in the last hospital visit? Knowledge: {KNOWLEDGE} Solution: {SOLUTION}	Question: Count the number of times that patient 54825 received a rt/left heart card cath procedure last year. Knowledge: {KNOWLEDGE} Solution: {SOLUTION}

Figure 9: Case study of long-term memory in EHRAgent on MIMIC-III dataset. From the original few-shot examples on the left, none of the questions related to either 'count the number' scenarios or procedure knowledge. In contrast, when we retrieve examples from the long-term memory, the new set is exclusively related to 'count the number' questions, thus providing a similar solution logic for reference.

Question: what was the amount of pioglitazone that patient 16992 were prescribed in total?

ReAct	EHRAgent

...
Observation 3: The corresponding HADM_ID are 155667, 107483.
Thought 4: We need to find the related information from prescriptions database.
Action 4: LoadDB[prescriptions]
Observation 4: We have successfully loaded the prescriptions database, including the following columns: Please modify it. ROW_ID, SUBJECT_ID, HADM_ID, STARTDATE, ENDDATE, DRUG, DOSE_VAL_RX, DOSE_UNIT_RX, ROUTE
Thought 5: Then, we need to filter information related to HADM_ID=155667 and DRUG=pioglitazone.
Action 5: FilterDB[HADM_ID=155667, DRUG=pioglitazone]
Observation 5: We have successfully filtered the data (1 rows).
Thought 6: Then we need to get the DOSE_VAL_RX value.
Action 6: FilterDB[DOSE_VAL_RX, sum]
Observation 6: The corresponding DOSE_VAL_RX value is 6.0.
Thought 7: We need to find the related information from prescriptions database.
Action 7: LoadDB[prescriptions]
Observation 7: We have successfully loaded the prescriptions database, including the following columns: Please modify it. ROW_ID, SUBJECT_ID, HADM_ID, STARTDATE, ENDDATE, DRUG, DOSE_VAL_RX, DOSE_UNIT_RX, ROUTE
Thought 8: Then, we need to filter information related to HADM_ID=155667 and DRUG=pioglitazone.
Action 8: FilterDB[HADM_ID=107483, DRUG=pioglitazone]
Observation 8: We have successfully filtered the data (1 rows).
Thought 9: Then we need to get the DOSE_VAL_RX value.
Action 9: FilterDB[DOSE_VAL_RX, sum]
Observation 9: The corresponding DOSE_VAL_RX value is 9.0.
...
	``
	# We can find the visiting information of patient 16992 in the admissions database.
	patient_db = LoadDB('admissions')
	filtered_patient_db = FilterDB(patient_db, 'SUBJECT_ID=16992')
	print(len(filtered_patient_db))
	hadm_id_list = GetValue(filtered_patient_db, 'HADM_ID, list')
	print(hadm_id_list)
	
	# As pioglitazone is a drug, we can find the corresponding information in the prescriptions database.
	prescriptions_db = LoadDB('prescriptions')
	
	# We can find the total amount of dose of pioglitazone that patient 16992 were prescribed by summing up the DOSE_VAL_RX in the prescriptions database.
	total_dose = 0
	for hadm_id in hadm_id_list:
	    filtered_prescriptions_db = FilterDB(prescriptions_db, 'HADM_ID={}||DRUG=pioglitazone'.format(hadm_id))
	    dose_list = GetValue(filtered_prescriptions_db, 'DOSE_VAL_RX, list')
	    print(dose_list)
	    for dose in dose_list:
	        total_dose += float(dose)
	
	answer = total_dose
	``

Figure 10: Case study 1 of code interface in EHRAgent on MIMIC-III dataset. The baseline approach, ReAct, lacks a code interface and encounters limitations when performing identical operations on multiple sets of data. It resorts to generating repetitive action steps iteratively, leading to an extended solution trajectory that may exceed the context limitations. In contrast, EHRAgent leverages the advantages of code structures, such as the use of 'for loops', to address these challenges more efficiently and effectively. The steps marked in red on the left side indicate the repeated actions by ReAct, while the steps marked in green are the corresponding code snippets by EHRAgent. By comparing the length and number of steps, the code interface can help EHRAgent save much context space.

Question: Show me patient 28020's length of stay of the last hospital stay.

ReAct	EHRAgent

ReAct

Thought 1: First, we need to find the last hospital stay of patient 28020.
Action 1: LoadDB[admissions]
Observation 1: We have successfully loaded the admissions database, including the following columns: ROW_ID, SUBJECT_ID, HADM_ID, ADMITTIME, DISCHTIME, ADMISSION_TYPE, ADMISSION_LOCATION, DISCHARGE_LOCATION, INSURANCE, LANGUAGE, MARITAL_STATUS, ETHNICITY, AGE.
Thought 2: Next, we need to filter the data to only include records for patient 28020.
Action 2: FilterDB[SUBJECT_ID=28020, max(DISCHTIME)]
Observation 2: We have successfully filtered the data (1 rows).
Thought 3: Now we have the last hospital stay record for patient 28020. We need to find the length of stay for this stay.
Action 3: GetValue[DISCHTIME-ADMITTIME, duration]
Observation 3: The value you are querying does not exist. Please modify it.
Thought 4: It seems there was an error in step 3. Let's correct the mistake and try again.
Action 4: GetValue[DISCHTIME - ADMITTIME, Calendar]
Observation 4: The value you are querying does not exist. Please modify it.
Thought 5: It seems that the calculation of the length of stay is not possible using the given steps. I apologize for the inconvenience.
Action 5: Finish[None]
Observation 5: Answer is INCORRECT

EHRAgent

```python
# Import the datetime module
import datetime

# Load the admissions
databaseadmissions_db = LoadDB('admissions')

# Filter the database for patient 28020
filtered_admissions_db = FilterDB(admissions_db, 'SUBJECT_ID=28020')

# Get the ADMITTIME and DISCHTIME of the last hospital
stayadmit_disch_tuple = SQLInterpreter('SELECT ADMITTIME, DISCHTIME FROM admissions WHERE SUBJECT_ID=28020 ORDER BY ADMITTIME DESC LIMIT 1')

# Extract the ADMITTIME and DISCHTIME from the tuple
admit_time, disch_time = admit_disch_tuple[0]

# Convert the ADMITTIME and DISCHTIME to datetime
objectsadmit_time = datetime.datetime.strptime(admit_time, '%Y-%m-%d %H:%M:%S')
disch_time = datetime.datetime.strptime(disch_time, '%Y-%m-%d %H:%M:%S')

# Calculate the length of hospital stay
length_of_stay = disch_time - admit_time

# Store the answer
answer = length_of_stay.days
```

Figure 11: Case study 2 of code interface in EHRAgent on MIMIC-III dataset. When encountering challenges in tool use, ReAct will keep making trials and can be stuck in the modification process. On the other hand, with code interface, EHRAgent can take advantage of Python built-in functions to help with debugging and code modification.

Question: Was the rdw value of patient 273 second measured greater than that first measured on the last hospital visit?

AutoGen	EHRAgent

AutoGen

```
...
# As RDW is a laboratory test, we can find the corresponding
ITEMID in the d_labitems database.
d_labitems_db = LoadDB('d_labitems')
filtered_d_labitems_db = FilterDB(d_labitems_db, 'LABEL=RDW')
item_id = GetValue(filtered_d_labitems_db, 'ITEMID')
...
```

🐍 Error: The filtering query LABEL=RDW is incorrect.

```
d_labitems_db = LoadDB('d_labitems')
filtered_d_labitems_db = FilterDB(d_labitems_db, 'LABEL=Red cell distribution width (RDW)')
item_id = GetValue(filtered_d_labitems_db, 'ITEMID')
...
```

🐍 Error: The filtering query LABEL=Red cell distribution width (RDW) is incorrect.

```
...
d_labitems_db = LoadDB('d_labitems')
filtered_d_labitems_db = FilterDB(d_labitems_db, 'LABEL=RDW')
item_id = GetValue(filtered_d_labitems_db, 'ITEMID')
...
```

EHRAgent

```
...
# As RDW is a laboratory test, we can find the corresponding
ITEMID in the d_labitems database.
d_labitems_db = LoadDB('d_labitems')
filtered_d_labitems_db = FilterDB(d_labitems_db, 'LABEL=RDW')
item_id = GetValue(filtered_d_labitems_db, 'ITEMID')
...
```

🐍 Error: The filtering query LABEL=RDW is incorrect.

There is no RDW value in the column. Five example values in the column are rbc, cd7, i, cd8, rdw.
Potential Reasons: The most possible reason for the error is that the filter condition for the 'LABEL' column in the 'd_labitems' database is case-sensitive. The error message indicates that there is a 'rdw' value in the column, but not 'RDW'. Therefore, the filter condition 'LABEL="RDW"' does not match any records. The filter condition should be changed to 'LABEL="rdw"'.

```
...
d_labitems_db = LoadDB('d_labitems')
filtered_d_labitems_db = FilterDB(d_labitems_db, 'LABEL=rdw')
item_id = GetValue(filtered_d_labitems_db, 'ITEMID')
...
```

Figure 12: Comparative case study of the interactive coding process between AutoGen (*left*) and EHRAgent (*right*), where EHRAgent delves deeper into environmental feedback via debugging module to achieve plan refinement.

Question: what was the last length of icu stay of patient 28037?

```python
from datetime import datetime
# We can find the ICU stay information of patient 28037 in the icustays database.
icustays_db = LoadDB('icustays')
filtered_icustays_db = FilterDB(icustays_db, 'SUBJECT_ID=28037 AND max(INTIME)')
# The length of ICU stay can be calculated by the difference between INTIME and OUTTIME in the icustays database.
intime = GetValue(filtered_icustays_db, 'INTIME')
outtime = GetValue(filtered_icustays_db, 'OUTTIME')
# Calculate the number of days that have passed since the patient was admitted to the hospital
answer = Calculate('({} - {}) / 86400'.format(in_time, out_time))
```
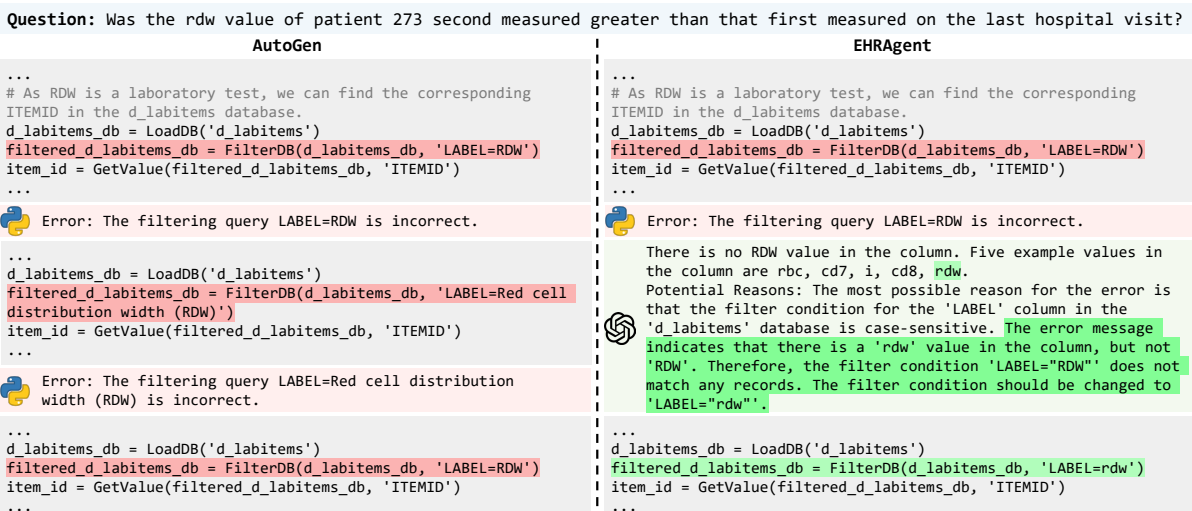
Error: The filtering query SUBJECT_ID=28037 AND max(INTIME) is incorrect. There is no 28037 AND max(INTIME) value in the column. Five example values in the column are 28037, 21037, 288037, 28039, 208037. Please check if you get the correct HADM_ID value. Please make modifications accordingly and make sure the rest code works well with the modification.

Potential Reasons: The most possible reason for the error is that the FilterDB function does not support multiple conditions in the CONDITIONS argument. The function is trying to find a single value "28037 AND max(INTIME)" in the SUBJECT_ID column, instead of filtering the database where HADM_ID equals 28037 and max(INTIME).

```python
from datetime import datetime
# We can find the ICU stay information of patient 28037 in the icustays database.
icustays_db = LoadDB('icustays')
filtered_icustays_db = FilterDB(icustays_db, 'SUBJECT_ID=28037||max(INTIME)')
# The length of ICU stay can be calculated by the difference between INTIME and OUTTIME in the icustays database.
intime = GetValue(filtered_icustays_db, 'INTIME')
outtime = GetValue(filtered_icustays_db, 'OUTTIME')
```

```python
# Calculate the number of days that have passed since the patient was admitted to the hospital
answer = Calculate('({} - {}) / 86400'.format(in_time, out_time))
```

Error: Invalid input query for Calculator. Please check the input query or use other functions to do the computation.. Please make modifications accordingly and make sure the rest code works well with the modification.

Potential Reasons: The most possible reason for the error is that the 'Calculate' function is trying to perform arithmetic operations on date values. The function is expecting numerical values for its calculation, but it's receiving date values from 'last_visit_date' and 'birth_date'.

```python
from datetime import datetime
# We can find the ICU stay information of patient 28037 in the icustays database.
icustays_db = LoadDB('icustays')
filtered_icustays_db = FilterDB(icustays_db, 'SUBJECT_ID=28037||max(INTIME)')
# The length of ICU stay can be calculated by the difference between INTIME and OUTTIME in the icustays database.
intime = GetValue(filtered_icustays_db, 'INTIME')
outtime = GetValue(filtered_icustays_db, 'OUTTIME')
intime = datetime.strptime(intime, '%Y-%m-%d %H:%M:%S')
outtime = datetime.strptime(outtime, '%Y-%m-%d %H:%M:%S')
length_of_stay = outtime - intime
if length_of_stay.seconds // 3600 > 12:
    answer = length_of_stay.days + 1
else:
    answer = length_of_stay.days
```

`4`

Output: The answer is 4.

Figure 13: A complete version of case study in Figure 7 showcasing interactive coding with environment feedback.

Question: count the number of patients who stayed in careunit csru until 2101?

Agent Prompt

Assume you have knowledge of following medical records: **[EHR Metadata (medical records descriptions)]**. Write a Python code to solve the given question. You can use the following functions: **[Tool Definitions (API name, API description)]**.

Medical Information Integration

Medical Information:
- The information about patients' stay in different care units can be found in the icustays database.
- Filter the records in the icustays database where FIRST_CAREUNIT or LAST_CAREUNIT is 'CSRU' and OUTTIME<=2101.
- The number of unique SUBJECT_IDs in these records will be the number of patients who stayed in careunit CSRU until 2101.

Demonstration Optimization through Long-Term Memory

Here are some examples:
Question: count patients who had a swab microbiology until 2104. Information: <med_info_1> Solution: <solution_1>
Question: count icu visits of patient 45612 until 2101. Information: <med_info_2> Solution: <solution_2>
Question: count patients who had a atg intake until 2 year ago. Information: <med_info_3> Solution: <solution_3>
Question: count patients who had a nutren pulmonary until 2103. Information: <med_info_4> Solution: <solution_4>

Interactive Coding

```python
# Import the datetime module
from datetime import datetime
# Load the icustays database
icustays_db = LoadDB('icustays')
# Filter the database for stays in CSRU
filtered_icustays_db = FilterDB(icustays_db, 'FIRST_CAREUNIT=csru OR LAST_CAREUNIT=csru')
# Get the SUBJECT_IDs and OUTTIMEs
subject_ids_and_outtimes = SQLInterpreter('SELECT SUBJECT_ID, OUTTIME FROM icustays WHERE FIRST_CAREUNIT="csru" OR LAST_CAREUNIT="csru"')
# Convert the OUTTIMEs to datetime objects and filter the SUBJECT_IDs for those until 2101
subject_ids_until_2101 = [subject_id for subject_id, outtime in subject_ids_and_outtimes if datetime.strptime(outtime, '%Y-%m-%d %H:%M:%S').year <= 2101]
# Convert the SUBJECT_IDs to a set to remove duplicates and count the number of unique SUBJECT_IDs
answer = len(set(subject_ids_until_2101))
```

Environmental Feedback

Error: The filtering query FIRST_CAREUNIT=csru OR LAST_CAREUNIT=csru is incorrect. There is no csru OR LAST_CAREUNIT=csru value in the column. Five example values in the column are csru, ccu, sicu, micu, tsicu. Please check if you get the correct FIRST_CAREUNIT value. Please make modifications accordingly and make sure the rest code works well with the modification.

Rubber Duck Debugging via Error Tracing

Potential Reasons: The most possible reason for the error is that the filter conditions in the FilterDB function are case-sensitive. The code is trying to filter with 'csru' while the actual value in the database is 'CSRU'. The correct filter condition should be 'FIRST_CAREUNIT=CSRU OR LAST_CAREUNIT=CSRU'.

```python
# Import the datetime module
from datetime import datetime
# Load the icustays database
icustays_db = LoadDB('icustays')
# Filter the database for stays in CSRU
filtered_icustays_db_first = FilterDB(icustays_db, 'FIRST_CAREUNIT=csru')
filtered_icustays_db_last = FilterDB(icustays_db, 'LAST_CAREUNIT=csru')
# Get the SUBJECT_IDs and OUTTIMEs
subject_ids_and_outtimes_first = SQLInterpreter('SELECT SUBJECT_ID, OUTTIME FROM icustays WHERE FIRST_CAREUNIT="csru"')
subject_ids_and_outtimes_last = SQLInterpreter('SELECT SUBJECT_ID, OUTTIME FROM icustays WHERE LAST_CAREUNIT="csru"')
# Combine the two lists
subject_ids_and_outtimes = subject_ids_and_outtimes_first + subject_ids_and_outtimes_last
# Convert the OUTTIMEs to datetime objects and filter the SUBJECT_IDs for those until 2101
subject_ids_until_2101 = [subject_id for subject_id, outtime in subject_ids_and_outtimes if datetime.strptime(outtime, '%Y-%m-%d %H:%M:%S').year <= 2101]
# Convert the SUBJECT_IDs to a set to remove duplicates and count the number of unique SUBJECT_IDs
answer = len(set(subject_ids_until_2101))
```

TypeError: strptime() argument 1 must be str, not None. The error messages occur in the code line "subject_ids_until_2101 = [subject_id for subject_id, outtime in subject_ids_and_outtimes if datetime.strptime(outtime, '%Y-%m-%d %H:%M:%S').year <= 2101]".

Potential Reasons: The most possible reason for the error is that the 'OUTTIME' value for some records in the 'icustays' database is None. The datetime.strptime() function is trying to convert a None value into a datetime object, which is causing the TypeError.

```python
# Import the datetime module
from datetime import datetime
# Load the icustays database
icustays_db = LoadDB('icustays')
# Filter the database for stays in CSRU
filtered_icustays_db_first = FilterDB(icustays_db, 'FIRST_CAREUNIT=csru')
filtered_icustays_db_last = FilterDB(icustays_db, 'LAST_CAREUNIT=csru')
# Get the SUBJECT_IDs and OUTTIMEs
subject_ids_and_outtimes_first = SQLInterpreter('SELECT SUBJECT_ID, OUTTIME FROM icustays WHERE FIRST_CAREUNIT="csru"')
subject_ids_and_outtimes_last = SQLInterpreter('SELECT SUBJECT_ID, OUTTIME FROM icustays WHERE LAST_CAREUNIT="csru"')
# Combine the two lists
subject_ids_and_outtimes = subject_ids_and_outtimes_first + subject_ids_and_outtimes_last
# Convert the OUTTIMEs to datetime objects and filter the SUBJECT_IDs for those until 2101
subject_ids_until_2101 = [subject_id for subject_id, outtime in subject_ids_and_outtimes if outtime is not None and datetime.strptime(outtime, '%Y-%m-%d %H:%M:%S').year <= 2101]
# Convert the SUBJECT_IDs to a set to remove duplicates and count the number of unique SUBJECT_IDs
answer = len(set(subject_ids_until_2101))
```

`39`

Output: The answer is 39.

Figure 14: Case study of the complete workflow in EHRAgent. With EHR metadata and tool definitions, EHRAgent (1) integrates medical information to locate the required tables/records, (2) retrieves relevant examples from long-term memory, (3) generates and executes code, (4) iteratively debugs with error messages until the final solution.