ЛАБОРАТОРНА РОБОТА № 1. РОЗРОБКА АВТОНОМНОГО AI-АГЕНТА З ДОВГОТРИВАЛОЮ ПАМ'ЯТТЮ НА БАЗІ SMOLAGENTS

Мета: набути практичних навичок у розробці та конфігурації автономних AI-агентів. Вивчити принципи наділення агента інструментами для виконання завдань. Опанувати техніку промпт-інжинірингу для керування поведінкою агента. Реалізувати механізм довготривалої пам'яті за допомогою архітектури Retrieval-Augmented Generation (RAG) з використанням векторних баз даних для вирішення проблеми обмеженого контекстного вікна великих мовних моделей.

Теоретичні відомості

Автономний AI-агент – це програмна система, ядром якої є велика мовна модель (LLM), що керує потоком виконання. На відміну від простого виклику LLM для генерації тексту, агент може ітеративно виконувати дії, аналізувати їх результати та планувати наступні кроки для досягнення складної мети. Для взаємодії з зовнішнім світом або виконання точних обчислень агенту надаються «інструменти» – звичайні функції коду, які LLM вчиться викликати. Фундаментальною проблемою при створенні агентів, здатних вести довгі діалоги, є обмежене контекстне вікно LLM. Технологія Retrieval-Augmented Generation (RAG) вирішує цю проблему, виносячи пам’ять у зовнішнє сховище. Кожен фрагмент інформації перетворюється на числовий вектор (ембединг), що відображає його семантичне значення. Коли агенту потрібна інформація з минулого, він шукає у цьому векторному сховищі найбільш релевантні спогади за змістом, а не за ключовими словами, і використовує лише їх, що робить пам’ять практично нескінченною та ефективною.

ЗАВДАННЯ ДЛЯ ВСІХ ВАРІАНТІВ

Необхідно розробити інтерактивного AI-асистента, який може виконувати наступні функції:

1. Взаємодіяти з користувачем у режимі командного рядка, приймаючи завдання у вигляді текстових запитів.
2. Використовувати набір інструментів для аналізу тексту: підрахунок слів та перевертання рядка.
3. Володіти довготривалою пам’яттю, яка дозволяє зберігати та згадувати факти з попередніх розмов. Пам’ять повинна бути реалізована за допомогою векторної бази даних faiss.
4. Агент повинен бути «активним» у керуванні своєю пам’яттю: самостійно приймати рішення про те, яку інформацію варто зберегти, використовуючи для цього спеціальний інструмент.
5. Агент повинен бути гнучким: вміти вибирати потрібний інструмент (або комбінацію інструментів) залежно від завдання користувача, а також вміти відповідати на прості запити, що не потребують інструментів.
6. Керування поведінкою агента має здійснюватися через детальний системний промпт, що використовує XML-теги для надійності та техніку few-shot prompting для навчання на прикладах.

ХІД РОБОТИ

Завдання 1. Налаштування середовища та встановлення залежностей.

Першим кроком є створення ізольованого віртуального середовища Python та встановлення всіх необхідних бібліотек, що забезпечить відтворюваність проєкту та дозволить уникнути конфлікту версій.

```
# Створення віртуального середовища
python -m venv .venv

# Активація середовища (Windows)
# .venv\Scripts\activate
# Активація середовища (macOS/Linux)
# source .venv/bin/activate

# Встановлення бібліотек
pip install "smolagents[litellm]" sentence-transformers faiss-cpu numpy
```

Цей блок команд створює папку *.venv*, що містить інтерпретатор Python та бібліотеки, ізольовані від системи. Команда *pip install* завантажує *smolagents* з екстра-пакетом *litellm* для роботи з локальними LLM через Ollama, *sentence-transformers* для створення векторних ембедингів, *faiss-cpu* для ефективного векторного пошуку на CPU, та *numpy* як залежність для *faiss*.

Завдання 2. Реалізація менеджера довготривалої пам'яті (*MemoryManager*).

Створюємо клас *MemoryManager*, який інкапсулює логіку роботи з RAG-пам'яттю. Він відповідає за перетворення тексту на вектори, їх збереження та швидкий семантичний пошук.

```
import faiss
from sentence_transformers import SentenceTransformer
import numpy as np
from smolagents import tool, CodeAgent, LiteLLMModel
from typing import Union
```

```
class MemoryManager:
    def __init__(self, model_name='all-MiniLM-L6-v2'):
        print("🧠 Ініціалізація Менеджера Пам'яті...")
        self.encoder = SentenceTransformer(model_name)
        self.dimension = self.encoder.get_sentence_embedding_dimension()
        self.index = faiss.IndexFlatL2(self.dimension)
        self.memories = []
        print("✅ Менеджер Пам'яті готовий.")

    def add_memory(self, text: str):
        print(f"📝 Збереження спогаду: '{text[:70]}...'")
        vector = self.encoder.encode([text])
        self.index.add(vector)
        self.memories.append(text)

    def retrieve_relevant_memory(self, query: str, k: int = 3) -> str:
        if not self.memories:
            return ""
        if self.index.ntotal < k:
            k = self.index.ntotal

        if k == 0: return ""

        print(f"🔍 Пошук {k} релевантних спогадів для запиту: '{query}'")
        query_vector = self.encoder.encode([query])
        distances, indices = self.index.search(query_vector, k)
        relevant_memories = [self.memories[i] for i in indices[0]]
        context = "\n".join(relevant_memories)
        print(f"💡 Знайдено релевантний контекст:\n--- КОНТЕКСТ ---
\n{context}\n--- КІНЕЦЬ КОНТЕКСТУ ---")
        return context

memory_manager = MemoryManager()
```

Клас *MemoryManager* при ініціалізації завантажує модель *all-MiniLM-L6-v2* для створення ембедингів та створює індекс *faiss.IndexFlatL2* для їх зберігання. Метод *add_memory* кодує текстовий спогад у вектор і додає його до індексу. Метод *retrieve_relevant_memory* кодує вхідний запит і шукає в індексі *k* найближчих (за евклідовою відстанню) векторів, повертаючи відповідні їм оригінальні текстові спогади.

Завдання 3. Створення інструментів для агента.

Визначаємо набір функцій, які агент зможе викликати для виконання завдань: інструменти для аналізу тексту, «мета-інструменти» для взаємодії з пам'яттю.

```
@tool
def count_words(text: str) -> int:
    """
    Counts the number of words in a given text string.
    Args:
        text: The text string to analyze.
    """
    print(f"Інструмент 'count_words' отримав текст: '{text}'")
    words = text.split()
    return len(words)

@tool
def reverse_text(text: str) -> str:
    """
    Reverses a given text string.
    Args:
        text: The text string to reverse.
    """
    print(f"Інструмент 'reverse_text' отримав текст: '{text}'")
    return text[::-1]

@tool
def save_fact_to_memory(fact_description: str, fact_content: str) -> str:
    """
    Saves an important fact or a result from another tool to your long-term
memory.
    Args:
        fact_description: A short description of what the fact is (e.g.,
'user's favorite color').
        fact_content: The actual content or value of the fact (e.g., 'blue').
    """
    memory_to_add = f"Fact about '{fact_description}': '{fact_content}'"
    memory_manager.add_memory(memory_to_add)
    return "Fact saved successfully."
@tool
def answer_from_context(answer_text: str) -> str:
    """
    Use this tool when the answer is already known from the provided context
or memory.
    Args:
        answer_text: The final, human-readable answer string.
    """
    return answer_text
```

Декоратор *@tool* з бібліотеки *smolagents* автоматично перетворює функції Python на інструменти, зрозумілі для LLM, шляхом аналізу їхніх docstrings та тайп-хінтів. *count_words* та *reverse_text* є простими аналітичними інструментами. *save_fact_to_memory* – це ключовий інструмент, що надає агенту можливість свідомо зберігати інформацію, викликаючи метод *add_memory* нашого *memory_manager*. *answer_from_context* слугує як дія-заглушка для ситуацій, коли відповідь вже відома з контексту.

Завдання 4. Конфігурація агента та системного промпта.

Це найважливіший етап, де ми визначаємо «особистість», правила та поведінку агента. Ми створюємо детальний системний промпт та ініціалізуємо *CodeAgent*.

```
model = LiteLLMModel(model_id="ollama/llama3:8b")
instructions_for_llama3 = """
You are an expert Python assistant with a long-term memory. You must respond
using an XML format with <thought> and <code> tags.

Follow these rules strictly:
1.  **Think in XML:** Always start by explaining your plan inside `<thought>`
and `</thought>` tags.
2.  **Use Memory Critically:** Analyze the provided context. If it DIRECTLY
answers the question, use `answer_from_context`. If the context is irrelevant
or doesn't help, state that you cannot find the answer.
3.  **Save Facts:** If you calculate a new result, save it using
`save_fact_to_memory`.
4.  **Write Code in XML:** Provide your Python script inside `<code>` and
`</code>` tags.
5.  **Call `final_answer()`:** The last line of your code MUST be a call to
`final_answer()`.

--- EXAMPLES ---

--- Example 1: Calculating and saving ---
Task: "Count the words in 'this is a test' and remember the count."
<thought>
I will use `count_words`, then `save_fact_to_memory`, and finally call
`final_answer`.
</thought>
<code>
text_to_process = 'this is a test'
word_count_result = count_words(text_to_process)
save_fact_to_memory(fact_description='word count for "this is a test"',
fact_content=str(word_count_result))
final_answer(f"I counted {word_count_result} words and saved this fact to
memory.")
</code>
```

```
--- Example 2: Answering directly and correctly from memory ---
Context: "Fact about 'Reversed word "Кіт"': 'тіК'\nFact about 'user's
favorite color': 'blue'"
Current Task: "What is my favorite color?"
<thought>
The context contains several facts. I need to find the one relevant to
'favorite color'. The fact "Fact about 'user's favorite color': 'blue'" is
the correct one. I will use `answer_from_context` to state this fact.
</thought>
<code>
known_fact = answer_from_context(answer_text="Based on my memory, your
favorite color is blue.")
final_answer(known_fact)
</code>

--- Example 3: Using only the `count_words` tool ---
Task: "How many words are in the phrase 'the quick brown fox'?"

Your generated code should look EXACTLY like this:
<code>
text_to_process = 'the quick brown fox'
word_count = count_words(text_to_process)
final_answer(f"There are {word_count} words in the phrase.")
</code>

--- Example 4: Using only the reverse_text tool ---
Task: "Reverse the word 'desserts'"
Your generated code should look EXACTLY like this:
<code>
original_text = 'desserts'
reversed_word = reverse_text(original_text)
final_answer(f"The reverse of 'desserts' is '{reversed_word}'.")
</code>

--- Example 5: Using BOTH tools ---
Task: "Analyze 'hello world': count its words and reverse it."
Your generated code should look EXACTLY like this:
<code>
text_to_process = 'hello world'
word_count_result = count_words(text_to_process)
reversed_text_result = reverse_text(text_to_process)
final_answer(f"Analysis complete. Word count: {word_count_result}. Reversed
text: '{reversed_text_result}'.")
</code>

Now, solve the user's current task following this XML format.
"""
```

```
agent = CodeAgent(
    tools=[count_words, reverse_text, save_fact_to_memory,
answer_from_context],
    model=model,
    instructions=instructions_for_llama3,
    code_block_tags=("<code>", "</code>")
)
```

Ми ініціалізуємо *LiteLLMModel* для роботи з локальною моделлю *llama3:8b* через Ollama. Системний промпт *instructions_for_llama3* використовує техніку few-shot prompting, надаючи моделі декілька прикладів правильної поведінки для різних сценаріїв. Використання XML-тегів (<*thought*>, <*code*>) підвищує надійність парсингу відповіді LLM. При створенні *CodeAgent* ми передаємо йому список доступних інструментів, сконфігуровану модель, наш детальний промпт, а також параметр *code_block_tags*, який вказує парсеру, де шукати код для виконання.

Завдання 5. Реалізація головного циклу оркестрації.

Створюємо інтерактивний цикл, який керує взаємодією між користувачем, менеджером пам’яті та агентом.

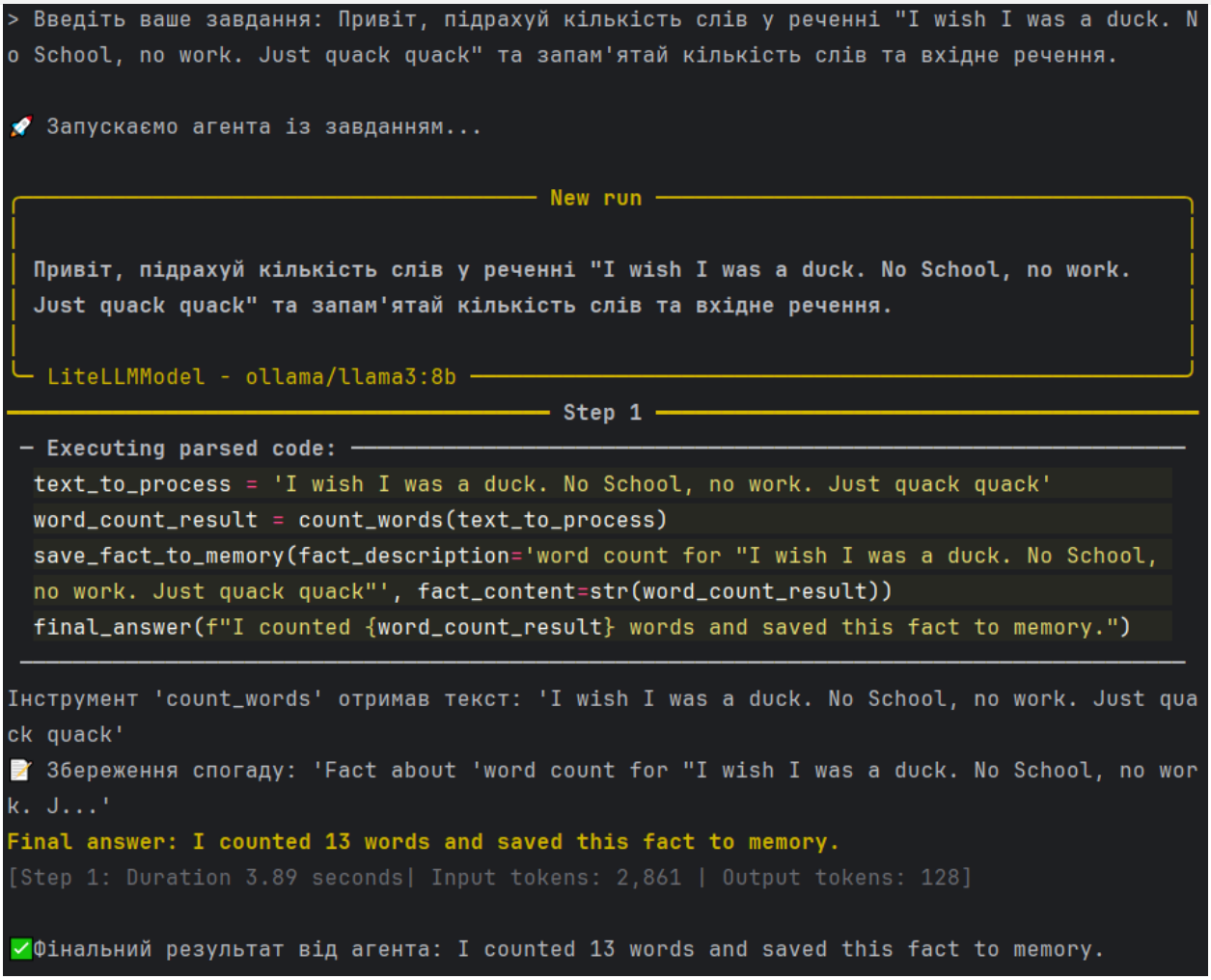
```
print("\n--- Інтерактивний режим з Активною RAG-пам'яттю ---")
print("Щоб почати новий чат, введіть '/new'. Щоб вийти, введіть 'exit'.")

while True:
    task = input("\n> Введіть ваше завдання: ")
    if task.lower() in ['exit', 'вихід']: break
    if task.lower() == '/new':
        memory_manager = MemoryManager()
        print("\n🧹 Пам'ять очищено. Починаємо новий чат.")
        continue
    retrieved_context = memory_manager.retrieve_relevant_memory(task)
    contextual_task = task
    if retrieved_context:
        contextual_task = f"Here is relevant context from our past
conversation:\n---\n{retrieved_context}\n---\n\nNow, using this context,
solve the following task: {task}"
    print(f"\n🚀 Запускаємо агента із завданням...\n")
    try:
        result = agent.run(contextual_task, reset=True)
        print(f"\n✅ Фінальний результат від агента: {result}")
    except Exception as e:
        print(f"\n❌ Під час виконання сталася помилка: {e}")
```

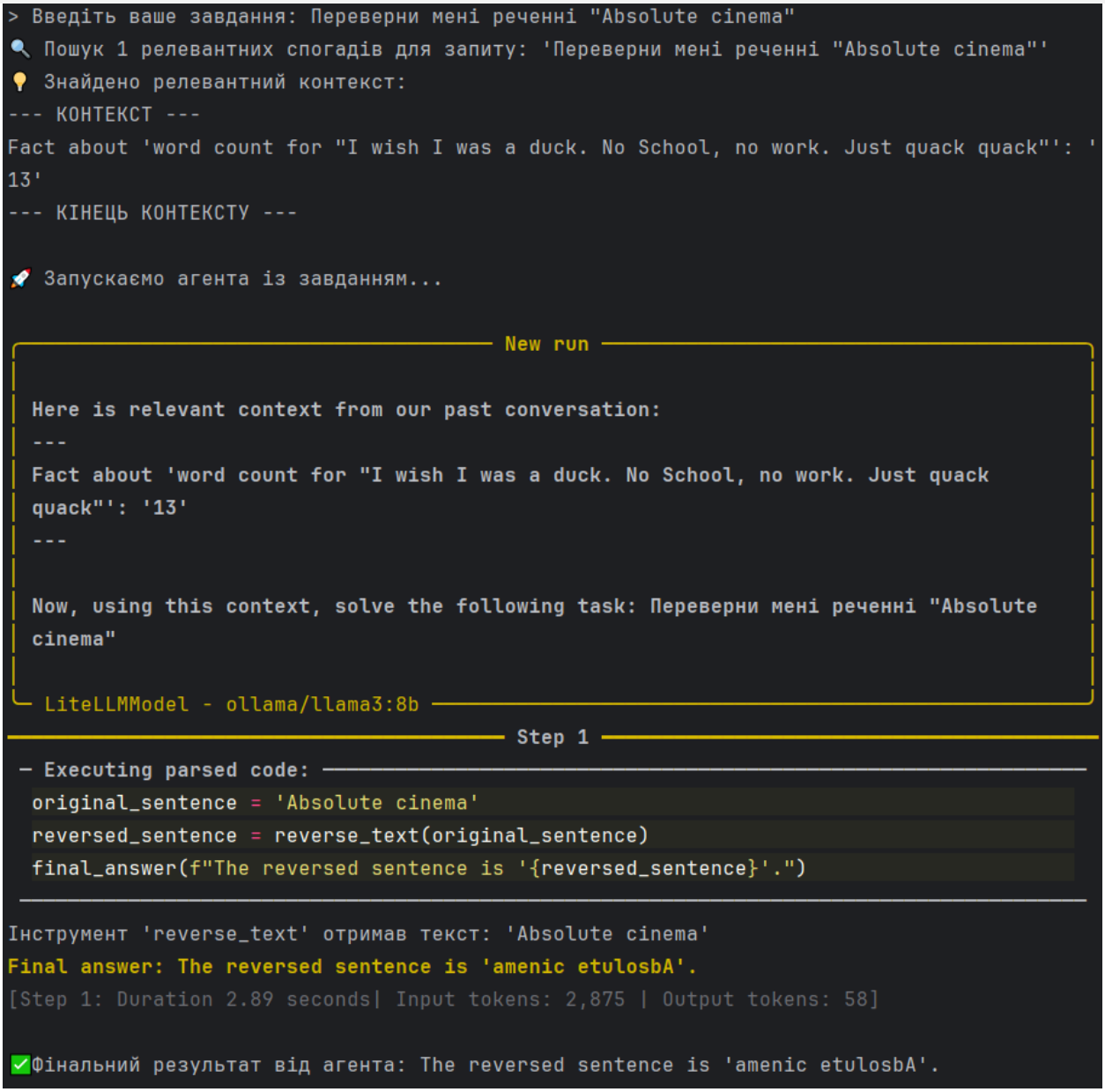
Цикл *while True* організовує сесію чату. Перед кожним викликом агента він звертається до *memory_manager* для пошуку релевантного контексту (фаза Retrieval). Потім він формує розширене завдання *contextual_task*, додаючи знайдений контекст до поточного запиту користувача (фаза Augmentation). Агент викликається з *reset=True*, оскільки довготривала пам'ять керується зовнішньою RAG-системою, і нам не потрібна внутрішня пам'ять *smolagents*. Після виконання завдання цикл очікує на новий ввід від користувача.

ПРИКЛАД РОБОТИ АГЕНТА

Промпт: Привіт, підрахуй кількість слів у реченні "I wish I was a duck. No School, no work. Just quack quack" та запам'ятай кількість слів та вхідне речення.



Промпт: Переверни мені реченні "Absolute cinema"



Промпт: Напиши факт, який ти пам'ятаєш

```
> Введіть ваше завдання: Напиши факт, який ти пам'ятаєш
🔍 Пошук 2 релевантних спогадів для запиту: 'Напиши факт, який ти пам'ятаєш'
💡 Знайдено релевантний контекст:
--- КОНТЕКСТ ---
Fact about 'word count for "I wish I was a duck"': '13'
Fact about 'word count for "I wish I was a duck. No School, no work. Just quack quack"': '
13'
--- КІНЕЦЬ КОНТЕКСТУ ---

🚀 Запускаємо агента із завданням...

╭──────────────────────────────────────── New run ─────────────────────────────────────────╮
│                                                                                          │
│ Here is relevant context from our past conversation:                                     │
│ ---                                                                                      │
│ Fact about 'word count for "I wish I was a duck"': '13'                                  │
│ Fact about 'word count for "I wish I was a duck. No School, no work. Just quack          │
│ quack"': '13'                                                                            │
│ ---                                                                                      │
│                                                                                          │
│ Now, using this context, solve the following task: Напиши факт, який ти пам'ятаєш       │
│                                                                                          │
╰─ LiteLLMModel - ollama/llama3:8b ────────────────────────────────────────────────────────╯
──────────────────────────────────────── Step 1 ─────────────────────────────────────────
 ─ Executing parsed code: ──────────────────────────────────────────────────────────────
  known_fact = answer_from_context(answer_text="Based on my memory, one of the facts I
  remember is that 'I wish I was a duck' has 13 words.")
  final_answer(known_fact)
 ──────────────────────────────────────────────────────────────────────────────────────
Final answer: Based on my memory, one of the facts I remember is that 'I wish I was a
duck' has 13 words.
[Step 1: Duration 3.78 seconds| Input tokens: 2,896 | Output tokens: 127]

✅Фінальний результат від агента: Based on my memory, one of the facts I remember is thatt
 'I wish I was a duck' has 13 words.
```

ВАРІАНТИ ІНДИВІДУАЛЬНИХ ЗАВДАНЬ

Загальне завдання для всіх варіантів

Розробити інтерактивного AI-асистента згідно з вашим варіантом. Асистент повинен мати довготривалу RAG-пам'ять та вміти свідомо зберігати в неї результати роботи своїх інструментів. Системний промпт повинен бути адаптований для коректного вибору та використання наданих інструментів, а також для виконання ключової вимоги.

Варіант	Найменування	Інструменти	Ключова вимога
1	Текстовий Аналітик	*count_sentences(text: str) -> int* Рахує кількість речень у тексті *find_longest_word(text: str) -> str* Знаходить найдовше слово у тексті	Агент повинен завжди надавати відповідь у форматі Markdown-таблиці
2	Математичний Помічник	*calculate_square_root(number: float) -> float* Обчислює квадратний корінь числа *calculate_factorial(number: int) -> int* Обчислює факторіал числа	Агент повинен завжди перевіряти вхідні дані (наприклад, не брати корінь з від'ємного числа) і повідомляти про помилку у разі некоректних даних
3	Генератор Паролів	*generate_password(length: int) -> str* Генерує випадковий пароль заданої довжини *check_password_strength(password: str) -> str* Оцінює надійність пароля («слабкий», «середній», «сильний»)	Після генерації пароля агент повинен автоматично перевіряти його надійність і повідомляти користувачу обидва результати
4	Перекладач Морзе	*text_to_morse(text: str) -> str* Конвертує текст в азбуку Морзе *morse_to_text(morse_code: str) -> str* Конвертує азбуку Морзе в текст	Агент повинен чітко розрізняти, який тип конвертації потрібен користувачу, і ніколи не плутати інструменти
5	Конвертер Валют	*convert_usd_to_uah(amount: float) -> float* Конвертує долари в гривні (використовуйте фіксований курс, напр. 40.5) *convert_uah_to_usd(amount: float) -> float* Конвертує гривні в долар	Агент повинен завжди вказувати курс, за яким була проведена конвертація, у фінальній відповіді
6	Файловий Інспектор	*create_file_with_content(filename: str, content: str)* Створює файл із заданим текстом *get_file_size(filename: str) -> int* Повертає розмір файлу в байтах	Агент повинен вміти обробляти помилки (наприклад, якщо файл не знайдено) і повідомляти про них користувачу
7	Текстовий Шифратор	*encrypt_caesar(text: str, shift: int) -> str* Шифрує текст шифром Цезаря *decrypt_caesar(text: str, shift: int) -> str* Дешифрує текст	Агент повинен зберігати в пам'ять ключ (зсув), який використовувався для шифрування, щоб потім можна було його «згадати» для дешифрування
8	Погодний Інформатор	*get_temperature(city: str) -> float* Повертає випадкову температуру для міста (напр. *random.uniform(15.0, 25.0)*) *get_weather_condition(city: str) -> str* Повертає випадковий стан погоди («сонячно», «хмарно», «дощ»)	Агент повинен завжди об'єднувати інформацію з обох інструментів у одне зв'язне речення («У місті [назва] зараз [стан]...»)

9	Аналізатор URL	*extract_domain(url: str) -> str* Витягує доменне ім'я з URL *check_if_https(url: str) -> bool* Перевіряє, чи використовує URL протокол HTTPS.	Агент повинен надавати відповідь у вигляді JSON-рядка
10	Генератор Імен	*generate_random_name(gender: str) -> str* Генерує випадкове ім'я (чоловіче/жіноче) *get_name_initials(full_name: str) -> str* Повертає ініціали імені	Агент повинен вміти генерувати ім'я та одразу ж повертати його ініціали в одній відповіді
11	Калькулятор Знижок	*calculate_discount_price(price: float, discount_percent: float) -> float* Обрахунок знижки *calculate_tax(price: float, tax_percent: float) -> float* Обрахунок податку	Агент повинен вміти послідовно застосовувати знижку, а потім податок до нової ціни
12	Лічильник Калорій	*get_calories_per_100g(product: str) -> int* Підрахунок калорій на 100 грам (використовуйте словник з 3-4 продуктів) *calculate_total_calories(calories_per_100g: int, weight_grams: int) -> float* Підрахунок загальної кількість калорій	Агент повинен питати у користувача вагу продукту, якщо вона не вказана
13	Конвертер Одиниць	*convert_km_to_miles(km: float) -> float* Конвертація кілометрів у милі *convert_miles_to_km(miles: float) -> float* Конвертація миль у кілометри	Агент повинен завжди вказувати коефіцієнт конвертації (1 миля = 1.60934 км) у відповіді
14	Генератор Кольорів	*generate_random_hex_color() -> str* Генерація рандомного hex коду кольору *hex_to_rgb(hex_code: str) -> tuple* Переведення hex у rgb	Агент повинен зберігати в пам'ять згенерований HEX-код, щоб потім його можна було конвертувати в RGB
15	Лічильник Символів	*count_characters(text: str, with_spaces: bool) -> int* Підрахунок символів *count_vowels(text: str) -> int* Підрахунок кількості голосних	Агент повинен вміти аналізувати текст за кількома критеріями (кількість символів з пробілами і без, кількість голосних) і надавати звіт
16	Таймер	*set_timer(seconds: int) -> str* Встановлення таймеру *check_timer_status() -> str* Перевірка залишку часу таймеру	Агент повинен використовувати *time.sleep()* в інструменті *set_timer* і повідомляти про завершення
17	Генератор Цитат	get_random_quote_by_category(category: str) -> str Генерація рандомної цитати get_quote_author(quote: str) -> str Надання інформації про автора цитати	Агент повинен завжди повертати цитату разом з її автором
18	Географічний Довідник	*get_country_capital(country: str) -> str* Надання інформації про столицю країни *get_country_population(country: str) -> int* Надання інформації про населення країни	Агент повинен вміти порівнювати населення двох країн
19	Текстовий Нормалізатор	*text_to_lowercase(text: str) -> str* Переведння тексту в нижній регістр *remove_punctuation(text: str) -> str* Видалення пунктуації з тексту	Агент повинен застосовувати обидва інструменти послідовно для повної «очистки» тексту
20	Генератор QR-кодів	generate_qr_code(data: str, filename: str) Генерація QR-коду (використовуйте бібліотеку *qrcode*) *read_text_from_file(filename: str) -> str* Зчитування тексту з файлу	Агент повинен вміти згенерувати QR-код для тексту, зберегти його, а потім зчитати цей текст з файлу для перевірки

** Можна обрати будь-яку іншу тему та розвинути свої інструменти.*

ІНСТРУКЦІЇ З НАЛАШТУВАННЯ ТА ЗАПУСКУ

Встановлення Ollama

Відвідайте офіційний сайт ollama.com та завантажте інсталятор для вашої операційної системи. Дотримуйтесь інструкцій зі встановлення. Після встановлення Ollama працюватиме як фоновий сервіс.

Завантаження мовної моделі

Відкрийте термінал або командний рядок та виконайте наступну команду, щоб завантажити модель Llama 3 (8 мільярдів параметрів). Завантаження може зайняти деякий час.

Примітка! *Можна використати будь-яку модель*

```
ollama pull llama3:8b
```

Запуск проєкту

Переконайтесь, що ви виконали крок 1 з «Ходу роботи» (встановили всі Python-бібліотеки). Помістіть весь розроблений код в один файл з назвою main.py. Запустіть проєкт з терміналу, перебуваючи в активному віртуальному середовищі.

```
python main.py
```

При першому запуску бібліотека *sentence-transformers* автоматично завантажить модель *all-MiniLM-L6-v2* (це відбудеться лише один раз). Після ініціалізації ви побачите запрошення на введення завдання.

Welcome to Github Repository

github.com/F1-bot/smolagent_sample.