



Asynchronous and Synchronous Python Code: A Comparative Study of Code Generation by Large Language Models and Human Programmers

- Tanvi Hadgaonkar, Habone Mohamed



Introduction

- In the world of artificial intelligence, Large Language Models (LLMs) like GPT-4 are revolutionizing how we think about tasks such as code generation.
- One key challenge is ensuring these models not only write code that looks right but also works right, especially when the order of steps in the code is complex.



Objective

We utilize GPT-4 to produce a diverse set of Python scripts encompassing both asynchronous and synchronous patterns.

We use Python developers written scripts achieving the same functional goals.


The resulting codebases are evaluated through a series of metrics, including execution time, resource utilization.



Methodology

- We generated a broad range of prompts.
- We then conducted an empirical comparison of their execution order against scripts executed by Python.
- This helped us identify any differences in how LLMs and traditional compilers handle code execution.

```
"prompt_1": {  
  "description": "Write a Python program that creates an asynchronous function to print 'Python Exercises!' with a two second delay",  
  "type": "asynchronous"  
},  
"prompt_2": {  
  "description": "Write a Python program that creates three asynchronous functions and displays their respective names with different",  
  "type": "asynchronous"  
},  
"prompt_3": {  
  "description": "Write a Python program that creates an asyncio event loop and runs a coroutine that prints numbers from 1 to 7 with",  
  "type": "asynchronous"  
},  
"prompt_4": {  
  "description": "Write a Python program that implements a coroutine to fetch data from two different URLs simultaneously using the 'a",  
  "type": "asynchronous"  
},  
"prompt_5": {  
  "description": "Write a Python program that runs multiple asynchronous tasks concurrently using asyncio.gather() and measures the ti",  
  "type": "asynchronous"  
},  
"prompt_6": {  
  "description": "Write a Python program to create a coroutine that simulates a time-consuming task and use asyncio.CancelledError to",  
  "type": "asynchronous"  
},  
"prompt_7": {  
  "description": "Write a Python program that implements a timeout for an asynchronous operation using asyncio.wait_for()",  
  "type": "asynchronous"  
},  
"prompt_8": {  
  "description": "Write a Python program that uses asyncio queues to simulate a producer-consumer scenario with multiple producers and",  
  "type": "asynchronous"  
}
```



Prompt 1: Write a Python program that creates three asynchronous functions and displays their respective names with different delays (1 second, 2 seconds, and 3 seconds)

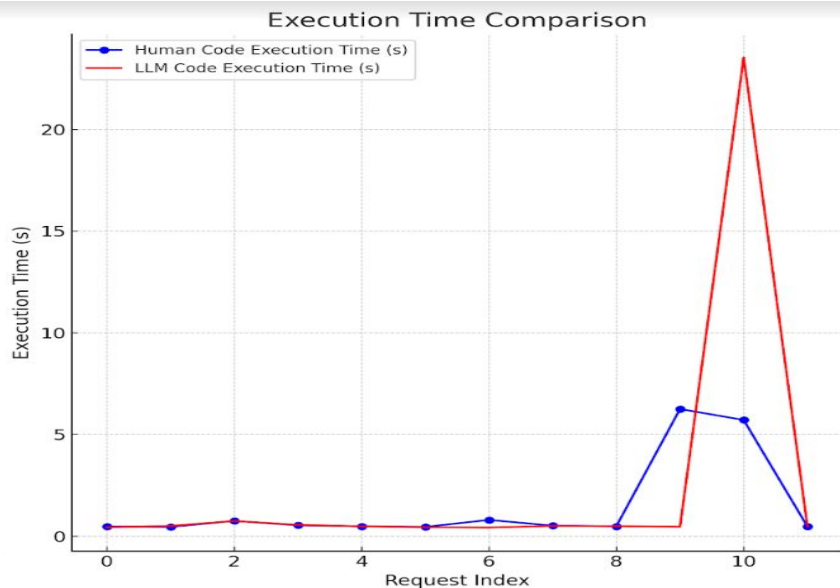
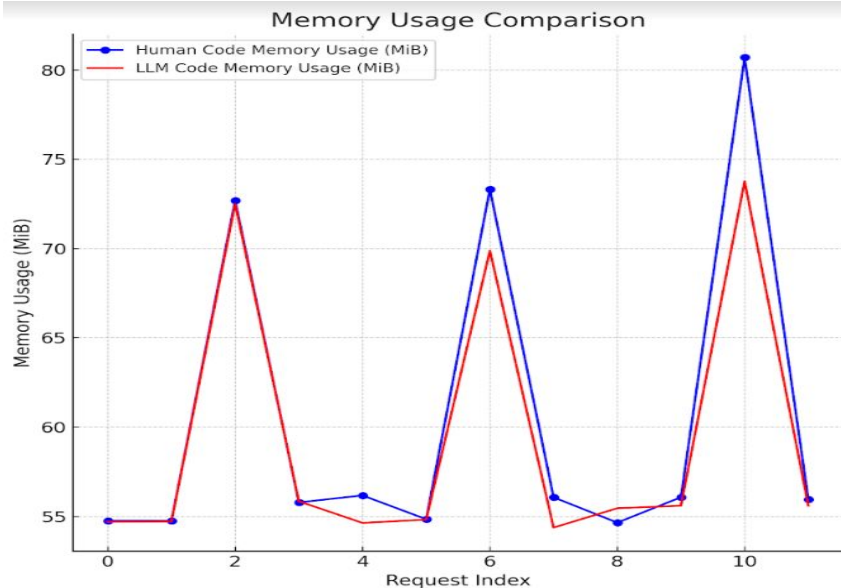
Asynchronous:

Code Generated by LLM.	Code Generated by human.
<pre> async def print_name_with_delay_1(): await asyncio.sleep(1) print("Function with 1 second delay") async def print_name_with_delay_2(): await asyncio.sleep(2) print("Function with 2 second delay") async def print_name_with_delay_3(): await asyncio.sleep(3) print("Function with 3 second delay") async def run_all(): await asyncio.gather(print_name_with_delay_1(), print_name_with_delay_2(), print_name_with_delay_3(),) asyncio.run(run_all()) </pre>	<pre> async def display_name_with_delay(name, delay): await asyncio.sleep(delay) print(name) async def main(): tasks = [display_name_with_delay("Asyn. function- 1", 1), display_name_with_delay("Asyn. function- 2", 2), display_name_with_delay("Asyn. function- 3", 3)] await asyncio.gather(*tasks) # Run the event loop asyncio.run(main()) </pre>
Output	Output
Function with 1 second delay Function with 2 second delay Function with 3 second delay	Asyn. function-1 Asyn. function-2 Asyn. function-3
Execution Time	Execution Time
2.997286081314087	3.0184617042541504
Memory Usage	Memory Usage
23.09375 MiB	23.1015625

Code Generated by LLM.	Code Generated by human.
<pre> import queue q = queue.Queue() # Check for empty queue if q.empty(): print("Queue is empty") q.put(1) q.put(2) q.put(3) # Process the queue sequentially while not q.empty(): # Dequeue a number from the queue num = q.get() # Count up to the number print("Counting up to", num) for i in range(1, num+1): print(i) print("Done counting up to", num) print("") # Check for empty queue if q.empty(): print("Queue is empty") </pre>	<pre> import queue def task(name, work_queue): if work_queue.empty(): print(f"Task {name} nothing to do") else: while not work_queue.empty(): count = work_queue.get() total = 0 print(f"Task {name} running") for x in range(count): total += 1 print(f"Task {name} total: {total}") def main(): work_queue = queue.Queue() for work in [15, 10, 5, 2]: work_queue.put(work) # Create some synchronous tasks tasks = [(task, "One", work_queue), (task, "Two", work_queue)] for t, n, q in tasks: t(n, q) if __name__ == "__main__": main() </pre>
Output	Output
Queue is empty Counting up to 1 1 Done counting up to 1 Counting up to 2 1 2 Done counting up to 2 Counting up to 3 1 2 3 Done counting up to 3 Queue is empty	Task One running Task One total: 15 Task One running Task One total: 10 Task One running Task One total: 5 Task One running Task One total: 2 Task Two nothing to do
Execution Time	Execution Time

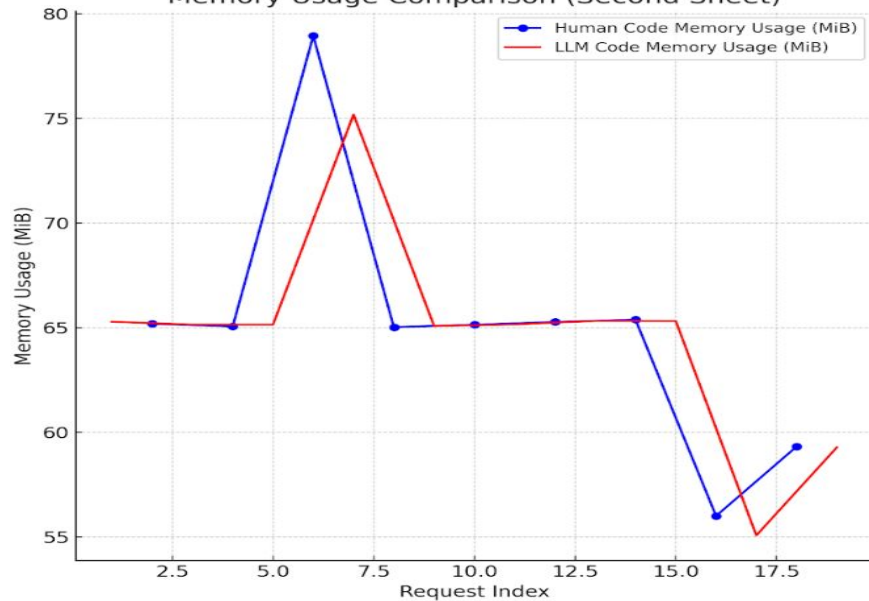
0.0005044937133789062	0.0015056133270263672
Memory Usage	Memory Usage
23.2578125	23.265625 MiB

Result-Synchronous

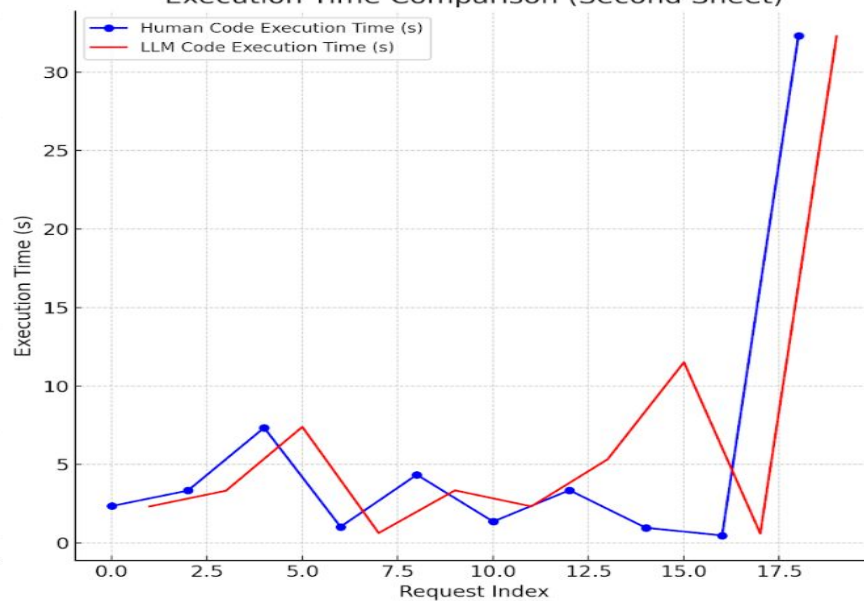


Result-Asynchronous

Memory Usage Comparison (Second Sheet)



Execution Time Comparison (Second Sheet)





Limitations

- The scope of our study is restricted by the modest number of samples, with only 13 instances of synchronous code and 12 for asynchronous code. This limitation is significant as it may affect the generalizability of our results..
- Despite AI's efficiency in execution, human-written code retains an advantage in memory consumption.
- TAI-generated asynchronous code execution order remains insufficiently explored. Additional scrutiny is required to confirm the reliability and predictability of asynchronous operations by AI.