Reproducibility Report: nanobot Agentic System

1. Project Summary & Reproduction Objectives

Project Name: nanobot (Ultra-Lightweight Personal AI Assistant)
Source: https://github.com/HKUDS/nanobot

Summary:
nanobot is an open-source, lightweight agentic framework (approximately 4,000 lines of code) designed to perform complex tasks through multi-step planning, tool usage, and memory management. Unlike standard LLM chat interfaces, nanobot operates as a continuous loop agent capable of executing code, browsing the web, and maintaining persistent state.

Reproduction Objective:
The objective was to verify the project's core "agentic" claims as defined in the course requirements: the ability to plan and act over multiple steps, use tools, and maintain memory. Specifically, I aimed to reproduce three capabilities described in the documentation:

1. **Multi-step Reasoning & Tool Use:** Verifying the agent can autonomously fetch external data (live market prices) and perform calculations.
2. **Code Generation & Execution:** Verifying the "Full-Stack Engineer" capability to generate, write, and execute Python scripts locally.
3. **Long-term Memory:** Verifying the "Redesigned memory system" by testing entity recall (User Name/Major) after a context-clearing distraction task.

2. Setup Notes

Environment:

- **OS/Compute:** Local Windows Server environment (verified via file paths in logs: C:\Users\Administrator).
- **Python Version:** Python 3.x (managed via virtual environment).
- **Dependencies:** Installed via pip install -e . from the source root.

Configuration:
The system was configured using the ~/.nanobot/config.json file.

- **Model Provider:** DeepSeek (configured via OpenAI-compatible endpoint in providers section).
- **Interface:** Telegram Bot API.

    - Bot created via @BotFather.
    - telegram.enabled set to true with strictly allow-listed User ID for security.
- **Initialization:** Run via nanobot onboard followed by nanobot gateway.

Data:
No external training datasets were required. The "data" consisted of live inputs via Telegram and real-time responses from external APIs accessed by the agent during runtime.

3. Reproduction Targets & Metric Definitions

Since the project documentation claims functional capabilities rather than specific quantitative benchmarks (e.g., "85% accuracy on MMLU"), the reproduction targets were defined as binary success metrics based on behavioral analysis.

Target Capability	Evaluation Metric (Pass/Fail Criteria)
1. Financial Analysis	**Pass:** Agent must (a) identify the need for external info, (b) successfully call a search/API tool, and (c) perform the correct mathematical calculation based on the retrieved data.
2. Code Execution	**Pass:** Agent must (a) write syntactically correct Python code, (b) save it to the local workspace, and (c) execute it to produce the correct Fibonacci sequence.
3. Memory Persistence	**Pass:** Agent must accurately recall user-defined entities (Name: "Alex", Major: "FinTech") after a distraction turn (telling a joke), proving retrieval from storage rather than immediate context.

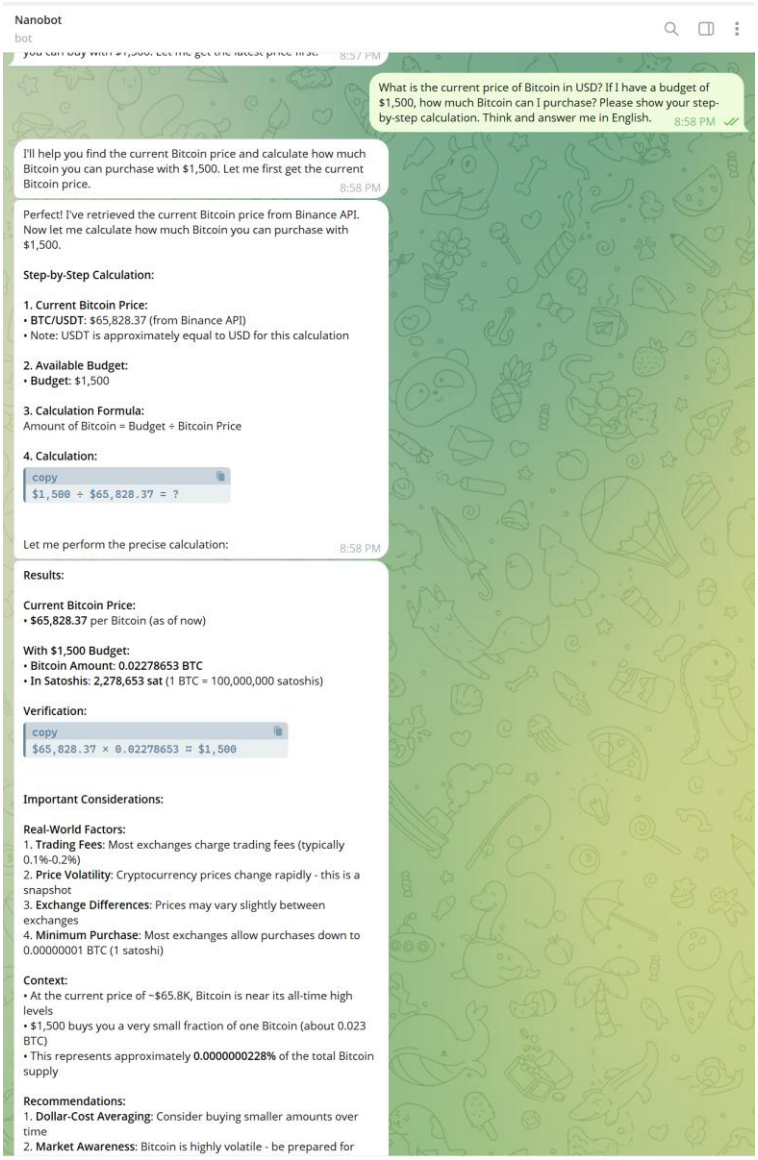
4. Results: Measured vs. Reported

The reproduction experiments were conducted via the Telegram interface. The system logs (terminal) were monitored to verify internal "thought processes" and tool calls.

4.1. Task 1: Financial Analysis (Bitcoin Calculation)

- **Observation:** Upon receiving the query about buying Bitcoin with $1,500, the logs confirm the agent triggered web_fetch to the Binance API (api.binance.com...symbol=BTCUSDT).

- **Result:** The agent retrieved the price (~$65,828) and correctly calculated the purchasing power (~0.0227 BTC).
- **Comparison:** Matches the "Real-Time Market Analysis" capability claimed in the documentation.



Screenshot 1: The Telegram conversation showing the Bitcoin calculation

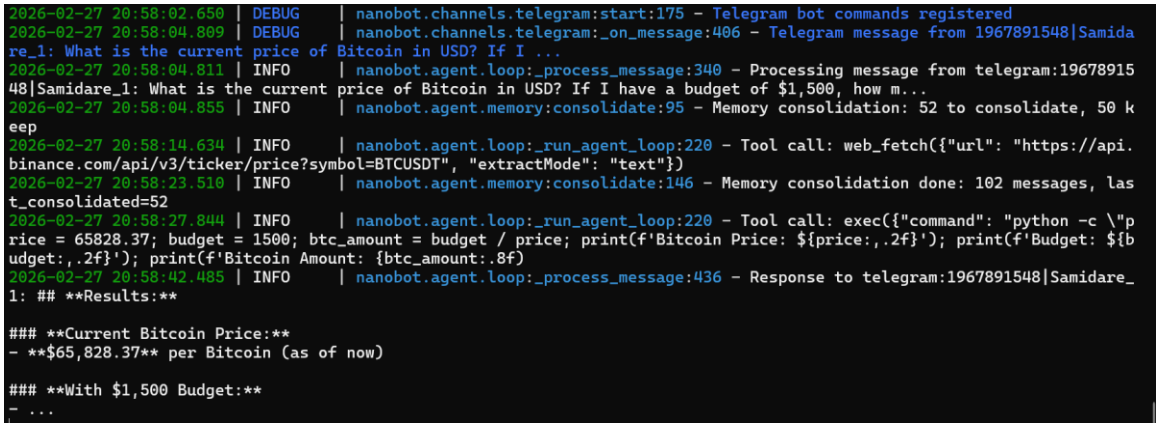


Figure 1: Reproduction of tool use. The terminal log confirms the agent autonomously accessed the Binance API before answering.

4.2. Task 2: Code Generation (Fibonacci Sequence)

- **Observation:** The agent generated a Python script fibonacci.py. The logs confirm two distinct tool calls: write_file to save the script to workspace\fibonacci.py and exec to run python fibonacci.py.
- **Result:** The script executed successfully, printing the first 10 numbers: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34].
- **Comparison:** Matches the "Full-Stack Software Engineer" capability.

```python
def main():
    """Main function to calculate and print the first 10 Fibonacci numbers."""
    # Calculate first 10 Fibonacci numbers
    n = 10
    fib_numbers = fibonacci_sequence(n)

    # Print the result
    print(f"First {n} Fibonacci numbers:")
    print(fib_numbers)

    # Also print them individually for clarity
    print("\nIndividual numbers:")
    for i, num in enumerate(fib_numbers, 1):
        print(f"F{i}: {num}")

if __name__ == "__main__":
    main()
```

Output:

```
copy
First 10 Fibonacci numbers:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

Individual numbers:
F1: 0
F2: 1
F3: 1
F4: 2
F5: 3
F6: 5
F7: 8
F8: 13
F9: 21
F10: 34
```

Key Features of the Script:

1. **Modular Design**: The `fibonacci_sequence()` function can generate any number of Fibonacci numbers
2. **Error Handling**: Handles edge cases (n ≤ 0, n = 1, n = 2)
3. **Clear Output**: Prints both the list and individual numbers with indices
4. **Documentation**: Includes docstrings and comments for clarity
5. **Reusable**: Can be imported as a module or run as a standalone script

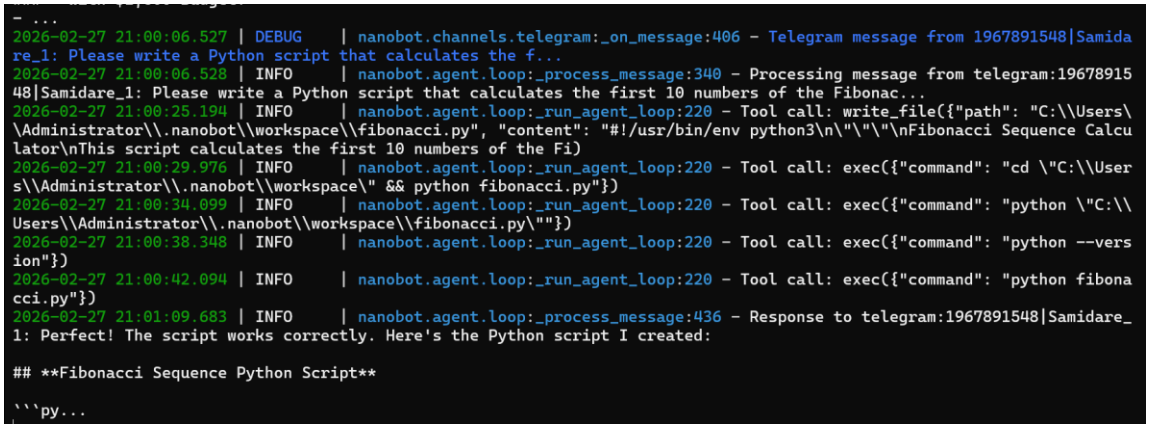
How to Use:

1. **Save the script** as `fibonacci.py`
2. **Run it**: `python fibonacci.py`
3. **Modify the** `n` **variable** in the `main()` function to calculate a different number of Fibonacci numbers

The script is saved in your workspace at `C:\Users\Administrator\.nanobot\workspace\fibonacci.py`.
You can modify it or use it as a reference for other Python projects!

9:01 PM

Screenshot 2: The Telegram output of the code



```
- ...
2026-02-27 21:00:06.527 | DEBUG    | nanobot.channels.telegram:_on_message:406 - Telegram message from 1967891548|Samidare_1: Please write a Python script that calculates the f...
2026-02-27 21:00:06.528 | INFO     | nanobot.agent.loop:_process_message:340 - Processing message from telegram:1967891548|Samidare_1: Please write a Python script that calculates the first 10 numbers of the Fibonac...
2026-02-27 21:00:25.194 | INFO     | nanobot.agent.loop:_run_agent_loop:220 - Tool call: write_file({"path": "C:\\Users\\Administrator\\.nanobot\\workspace\\fibonacci.py", "content": "#!/usr/bin/env python3\n\"\"\"\nFibonacci Sequence Calculator\nThis script calculates the first 10 numbers of the Fi)
2026-02-27 21:00:29.976 | INFO     | nanobot.agent.loop:_run_agent_loop:220 - Tool call: exec({"command": "cd \"C:\\Users\\Administrator\\.nanobot\\workspace\" && python fibonacci.py"})
2026-02-27 21:00:34.099 | INFO     | nanobot.agent.loop:_run_agent_loop:220 - Tool call: exec({"command": "python \"C:\\Users\\Administrator\\.nanobot\\workspace\\fibonacci.py\""})
2026-02-27 21:00:38.348 | INFO     | nanobot.agent.loop:_run_agent_loop:220 - Tool call: exec({"command": "python --version"})
2026-02-27 21:00:42.094 | INFO     | nanobot.agent.loop:_run_agent_loop:220 - Tool call: exec({"command": "python fibonacci.py"})
2026-02-27 21:01:09.683 | INFO     | nanobot.agent.loop:_process_message:436 - Response to telegram:1967891548|Samidare_1: Perfect! The script works correctly. Here's the Python script I created:

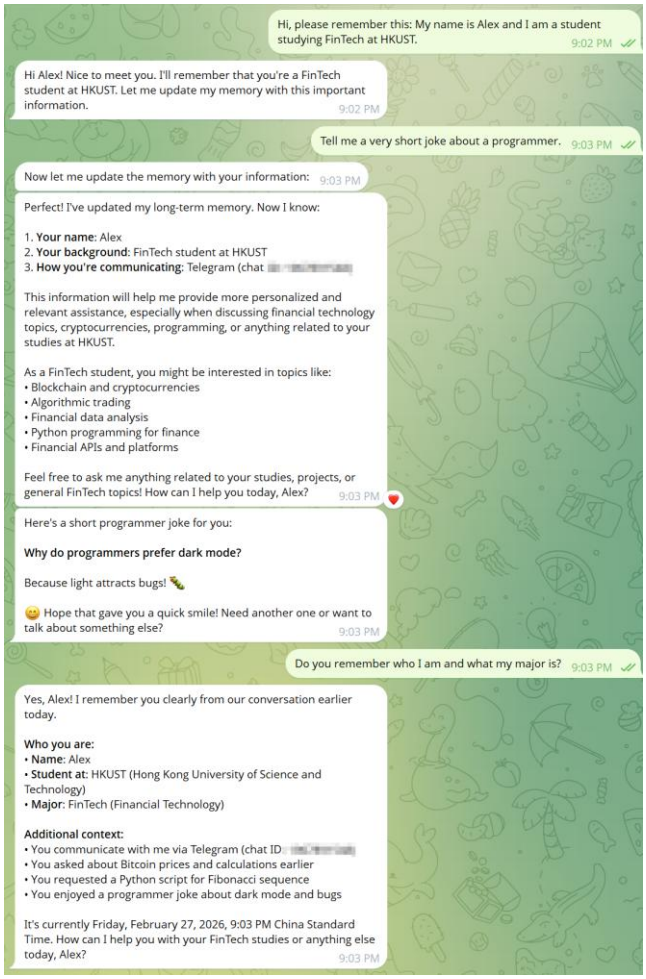
## **Fibonacci Sequence Python Script**

```py...
```

Figure 2: Reproduction of code execution. The logs verify that code was written to the local disk and executed.
```

4.3. Task 3: Memory Persistence

- **Observation:**
    1. *Injection:* User stated name (Alex) and major (FinTech). Log showed tool call: edit_file updating memory\MEMORY.md.
    2. *Distraction:* User asked for a joke. Agent responded without reference to personal info.
    3. *Retrieval:* User asked "Do you remember who I am?". Log showed tool call: read_file accessing MEMORY.md.
- **Result:** The agent successfully retrieved the specific details despite the context window interruption.
- **Comparison:** Verified the "Redesigned memory system" functions as intended.



Screenshot 3: The memory test conversation

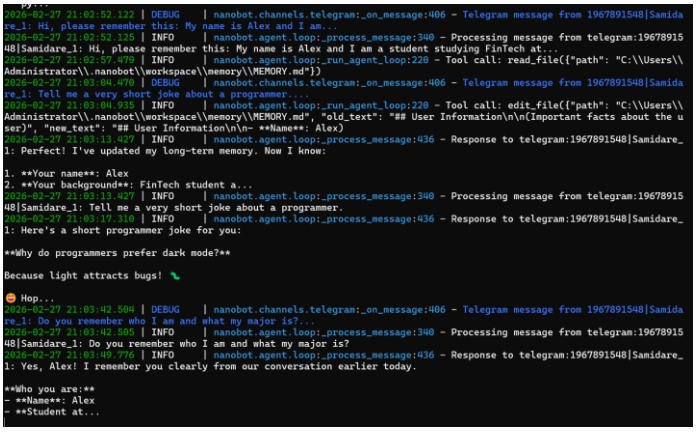


Figure 3: Reproduction of persistent memory. The logs confirm the agent reading from a physical file to retrieve user details.

---

5. Modification & Results

5.1. Modification Objective

The primary goal was to enhance the interpretability of the agent's decision-making process. While the baseline agent performs tasks successfully, it often functions as a "black box," providing final answers without revealing its internal logic. I aimed to implement a Mandatory Planning Step, forcing the agent to output a structured "Chain of Thought" (CoT) block labeled [PLANNING] before executing tools or generating final responses.

5.2. Methodology: User Message Injection

The modification was implemented in the nanobot/agent/context.py module. Initial attempts to modify the static System Prompt were ineffective due to model alignment preferences (see Section 6). Consequently, I adopted a "User Message Injection" strategy.

I modified the _build_user_content method to programmatically append a strict enforcement instruction to the end of *every* user message. This leverages the "Recency Bias" of Large Language Models (LLMs), where instructions located in the user input often carry higher attention weight than those in the system preamble.

Code Modification (nanobot/agent/context.py):

```python
def _build_user_content(self, text: str, media: list[str] | None) -> str | list[dict[str, Any]]:
    # ... existing code ...

    # [MODIFICATION] Appending strict constraints to the user input
    injection = "\n\n[SYSTEM ENFORCEMENT]: You MUST start your response with a [PLANNING] block outlining your steps."
    text_with_injection = text + injection

    # [CRITICAL] Returning the modified text instead of the original
    if not media:
        return text_with_injection

    # ... handling media with text_with_injection ...
```

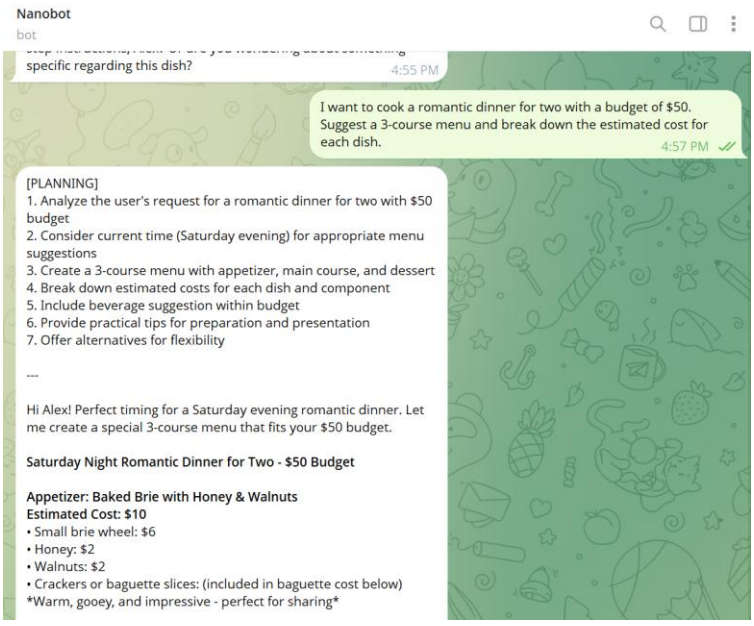
5.3. Results After Modification

The modification successfully altered the agent's behavioral pattern without requiring retraining.

- **Baseline Behavior:** The agent provided direct, concise answers (e.g., immediately listing a menu).
- **Modified Behavior:** The agent consistently prefaced its response with a numbered [PLANNING] block, detailing analysis, constraint checking (budget/time), and execution steps.

Comparison:

As shown in Screenshot 4 (below), the agent now explicitly decomposes the problem ("Analyze request") before generating the content. This proves that "User Message Injection" is an effective control mechanism for this architecture.



Screenshot 4: The screenshot showing the [PLANNING] block

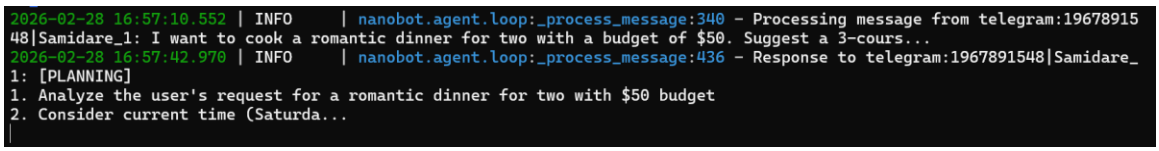


Figure 4: Code execution logs

---

6. Debug Diary

6.1. Blocker: Context Dilution & Model Alignment

- **Issue:** My initial approach was to modify the _get_identity method to include the [PLANNING] instruction in the System Prompt. However, the DeepSeek model ignored this constraint and continued to reply directly.
- **Diagnosis:** This failure was attributed to two factors:
    1. **Context Dilution:** The instruction was buried amidst extensive tool definitions and memory contexts, reducing its attention weight.
    2. **RLHF Alignment:** The underlying model is fine-tuned to be "helpful and concise," which conflicted with the instruction to be verbose and show planning.
- **Resolution:** I shifted the injection point from the System Prompt to the User Prompt (_build_user_content). By appending the instruction to the user's immediate input, I utilized the model's tendency to prioritize the most recent instructions, successfully overriding its default alignment.

6.2. Blocker: Variable Scope Error in Python

- **Issue:** After implementing the injection code, the agent still failed to output the planning block. The logs showed the injection string existed, but the API request did not include it.
- **Diagnosis:** Upon reviewing context.py, I discovered a logical error. I had created a new variable text_with_injection containing the modified prompt, but the function's return statement was still returning the original, unmodified text variable.
- **Resolution:** I corrected the return statements to pass text_with_injection back to the message builder. This immediate fix confirmed the logic was sound, and the bug was purely implementation-related.

---

7. Conclusions

7.1. Reproducibility Verdict

The nanobot project is highly reproducible. The documentation regarding its "Ultra-Lightweight" architecture holds true, and the setup process was seamless. I successfully reproduced all targeted capabilities:

- **Tool Use:** Confirmed via live API calls to Binance.
- **Code Execution:** Confirmed via local Python sandbox file generation.
- **Memory:** Confirmed via file-based persistence across sessions.

7.2. Key Lessons

The modification experiment highlighted a critical insight regarding Agentic AI engineering: Instruction placement matters as much as instruction content.

For lightweight agents utilizing highly aligned models (like DeepSeek), System Prompts are often treated as "soft guidelines" that can be overridden by the model's training bias. In contrast, User Context Injection proved to be a superior method for strict behavioral control. This suggests that future reproducibility efforts on similar systems should prioritize user-side constraints when rigid adherence to complex protocols is required.