Ordering Bot

A report describing the solution to the ordering bot challenge, as well as its evaluation.

Functional requirements

The following core capabilities have been identified as functional requirements for the Order Assistant:

- General chatbot interaction capabilities.
- Tracking the status of existing orders.
- Handling order cancellation requests.
- Strict adherence to the defined cancellation policy (e.g., eligibility limited to 10 days post-order placement).

Non-functional requirements

Key non-functional requirements guiding the design include:

- **Reliability and Safety:** Ensure order cancellations are reliable and safe, mitigating risks associated with potential LLM inaccuracies or hallucinations.
- **Cost-Effectiveness:** The solution should be reasonably economical, considering its dual function as a standard chatbot alongside order management tasks.
- **Modularity:** The system architecture must be modular to facilitate:
    - **Testability:** Enabling straightforward unit and integration testing.
    - **Extensibility:** Allowing for future enhancements and feature additions.

Out of scope

The following aspects are explicitly designated as out of scope for the current implementation:

- **Security:**
    - User authentication or access control mechanisms for the assistant itself.
    - Defense against adversarial attacks or prompt injection during user conversations.
- **Performance Testing:**
    - Load testing, stress testing, or concurrency testing to evaluate performance under high traffic conditions.

Reviewed concepts

Three high-level conceptual approaches were evaluated for the Order Assistant:

LLM with Direct Function Calling

- This approach pairs the Language Model (LLM) directly with tools (functions) using native function calling capabilities provided by inference services.
- **Advantages:** Implementation is straightforward, and it is relatively cost-effective due to fewer LLM calls per interaction.
- **Disadvantages:** Reliability poses a challenge, particularly concerning the risk of unintended actions like cancelling orders without explicit user intent, conflicting with non-functional safety requirements.

Separate NLU / NLG Instances

- This concept involves two distinct LLM instances: one for Natural Language Understanding (NLU) to interpret user intent and classify requests, and another for Natural Language Generation (NLG) to manage the conversation with the user.
- Function execution is triggered based solely on the NLU component's output, with the results then passed to the NLG component for formulating the response.
- **Advantages:** Offers enhanced safety as the conversational NLG model does not have direct access to execute functions.
- **Disadvantages:** Incurs higher operational costs due to requiring two LLM calls for each user query.

LLM with Functions and Human Approval

- This model adapts the direct function calling approach by incorporating a mandatory user confirmation step for sensitive actions, specifically order cancellation.
- Before executing a cancellation, the system prompts the user for explicit confirmation, typically via a UI element like a button, or potentially through SMS/email verification. It's crucial to note that in this design, the LLM tool (`_tool_cancel_order_check`) *only* verifies if an order is eligible for cancellation based on policy (e.g., age) by checking its details (via `GET /track/{id}`). It does *not* execute the cancellation itself. This separation is a core safety feature.
- **Advantages:** Guarantees operational safety by ensuring cancellations only occur with explicit user consent, while retaining the cost-efficiency of direct function calling for less sensitive operations like tracking or listing orders.
- **Disadvantages:** Introduces an extra step in the workflow for specific actions.

Design

The chosen design is "LLM with functions and human approval". This approach was selected as it effectively balances functional requirements with the non-functional priorities of reliability and cost-efficiency.

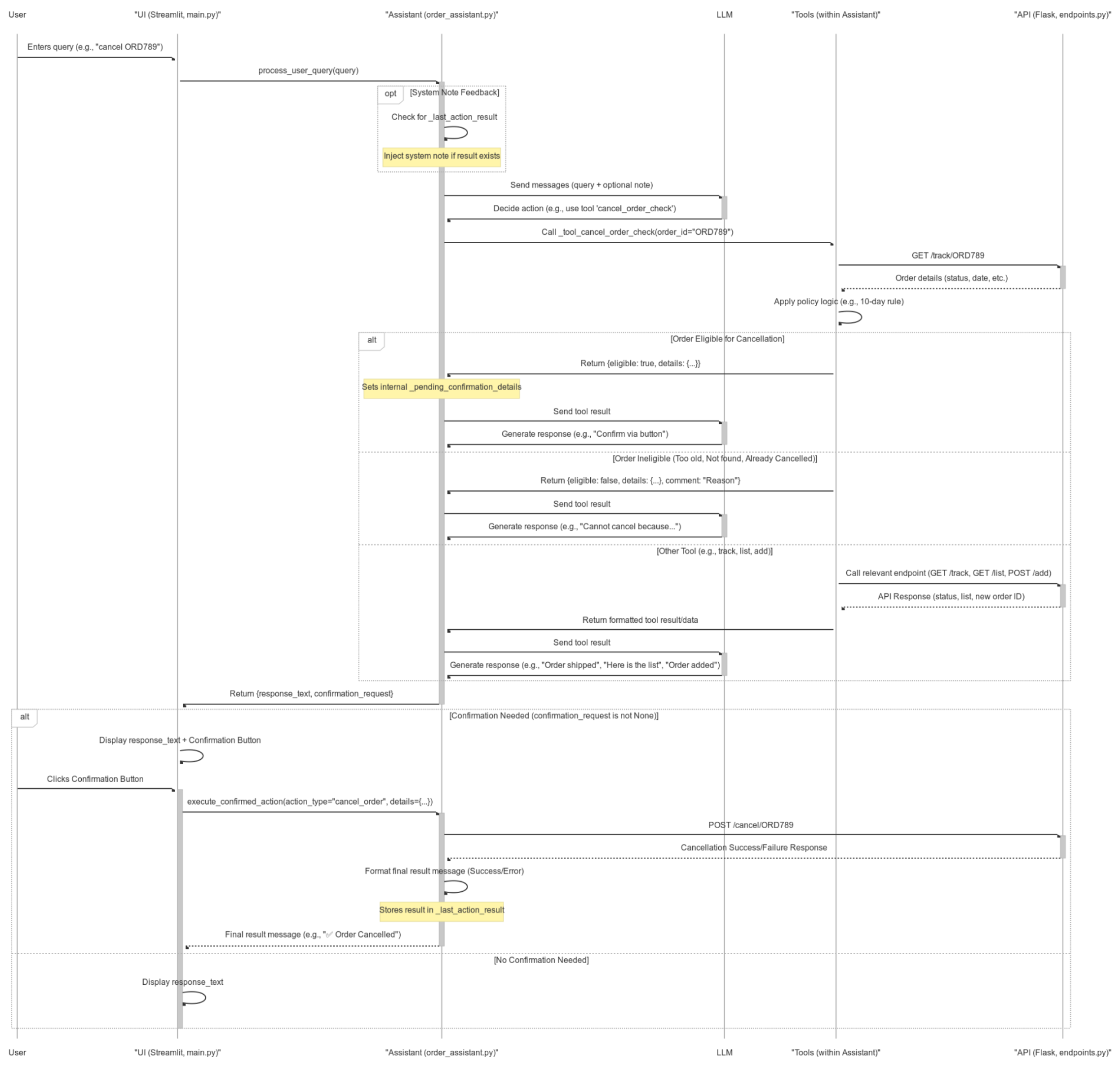
The system operates as follows:

- The LLM interprets user requests to determine the appropriate action (track, add, check cancellation).
- Specific tools linked to the LLM interact with the backend API for data retrieval and initial checks. The conceptual `OrderTracking` requirement is fulfilled by the agent calling the `GET /track/{order_id}` endpoint, and the `OrderCancellation` process involves checking eligibility via `GET /track/{order_id}` followed by executing the actual cancellation, after user confirmation, via the `POST /cancel/{order_id}` endpoint.
- For sensitive actions like cancellation, the _tool_cancel_order_check function verifies eligibility against the 10-day policy using API data.
- Crucially, if an order is eligible for cancellation, the assistant *does not* proceed automatically. Instead, it informs the user that explicit confirmation is required via the UI. This ensures that the potentially destructive action of cancellation is never triggered directly by the LLM. The actual `POST /cancel/{order_id}` API call, which performs the cancellation, is invoked separately by the `execute_confirmed_action` function only *after* receiving explicit confirmation from the user through the interface. This deliberate two-step flow prioritises operational safety.
- The actual cancellation API call is only executed after this external confirmation is received through the execute_confirmed_action flow.

This multi-step process, including the mandatory user confirmation for cancellation, ensures operational safety by preventing unintended actions while leveraging the efficiency of direct function calling for other tasks. The sequence diagram provides a detailed illustration of this interaction flow.

Sequence diagram

https://www.mermaidchart.com/raw/31a499ee-63d7-4bfb-8b98-36134d32b324?theme=light&version=v0.1&format=svg


User
"UI (Streamlit, main.py)"
"Assistant (order_assistant.py)"
LLM
"Tools (within Assistant)"
"API (Flask, endpoints.py)"
Enters query (e.g., "cancel ORD789")
process_user_query(query)
opt
[System Note Feedback]
Check for _last_action_result
Inject system note if result exists
Send messages (query + optional note)
Decide action (e.g., use tool 'cancel_order_check')
Call _tool_cancel_order_check(order_id="ORD789")
GET /track/ORD789
Order details (status, date, etc.)
Apply policy logic (e.g., 10-day rule)
alt
[Order Eligible for Cancellation]
Return {eligible: true, details: {...}}
Sets internal _pending_confirmation_details
Send tool result
Generate response (e.g., "Confirm via button")
[Order Ineligible (Too old, Not found, Already Cancelled)]
Return {eligible: false, details: {...}, comment: "Reason"}
Send tool result
Generate response (e.g., "Cannot cancel because...")
[Other Tool (e.g., track, list, add)]
Call relevant endpoint (GET /track, GET /list, POST /add)
API Response (status, list, new order ID)
Return formatted tool result/data
Send tool result
Generate response (e.g., "Order shipped", "Here is the list", "Order added")
Return {response_text, confirmation_request}
alt
[Confirmation Needed (confirmation_request is not None)]
Display response_text + Confirmation Button
Clicks Confirmation Button
execute_confirmed_action(action_type="cancel_order", details={...})
POST /cancel/ORD789
Cancellation Success/Failure Response
Format final result message (Success/Error)
Stores result in _last_action_result
Final result message (e.g., "✅ Order Cancelled")
[No Confirmation Needed]
Display response_text
User
"UI (Streamlit, main.py)"
"Assistant (order_assistant.py)"
LLM
"Tools (within Assistant)"
"API (Flask, endpoints.py)"

Evaluation and metrics

Testing flow

In the interest of simplicity, a testing framework will consist of Python logging with analysis of logged outputs. Normally, a more robust approach would need to be considered where observability, data storage and analysis are done by dedicated tools. The test case has also been simplified, normally, one would expect a 3-layered approach:

- Unit tests
- Unit tests with mocked model (fake LLM)
- Full run with an LLM

For our purposes, we will limit ourselves to a few test cases using an actual LLM.

Testing cases overview

The testing involved several scenarios targeting the core functionalities of the Order Assistant. These tests were designed to verify if the assistant correctly identified the user's intent, selected the appropriate tool, extracted the necessary parameters, and handled the API responses, including edge cases and policy checks. Key test cases included:

- Tracking existing orders (e.g., ORD123).
- Attempting to track non-existent orders (e.g., ORD999).
- Checking cancellation eligibility for:
    - An eligible order within the policy (e.g., ORD789), expecting confirmation request.
    - An ineligible order due to the 10-day policy (e.g., ORD456).
    - A non-existent order (e.g., ORD000).
    - Boundary cases exactly on the 10-day limit (e.g., ORD910), expecting confirmation request.
    - Boundary cases just outside the 11-day limit (e.g., ORD911).
    - An order already marked as cancelled (e.g., ORD912).
- Simulating a network fault during order tracking to test error handling.

The execution flow and specific tool calls for these cases were logged for analysis.

Skipped tests

While the implemented test suite provides a solid baseline for validating the agent's core tool usage and adherence to the cancellation policy, several categories of tests were skipped due to time and resource constraints, or were identified as areas for future improvement. A more comprehensive evaluation framework would ideally include:

- **LLM Determinism and Cost Efficiency**
    - The current tests rely on live LLM calls, which can introduce variability in results and incur costs. Introducing deterministic tests using model stubs (like a `FakeChatCompletionClient` returning pre-defined responses) would allow for faster, cheaper, and more predictable testing, suitable for CI/CD pipelines. Stochastic tests against the real model could then be run separately for regression checks.
- **End-to-End Confirmation Flow Execution**
    - The existing tests verify that the *request* for cancellation confirmation is correctly generated when an order is eligible. However, they don't automate the final step of simulating the user's confirmation click in the UI and verifying that the subsequent `execute_confirmed_action` function successfully calls the `POST /cancel` API endpoint and handles its response.

- **Comprehensive Edge Case and Negative Testing:**
    - **Concurrency:** No tests simulated simultaneous user queries to check for potential race conditions, particularly around the handling of pending confirmation states (`_pending_confirmation_details`).
    - **API Fault Tolerance:** Beyond the simulated timeout, tests could inject other API failures (e.g., 5xx errors, malformed responses, other 4xx errors besides 404) to ensure the agent handles them gracefully.
- **End-to-End Output Quality Assessment**
    - The current assertions check for basic keywords in the agent's response. They don't quantitatively score the overall quality of the natural language response (e.g., helpfulness, politeness, clarity of policy citation when needed), which could be assessed using LLM-based evaluators or more sophisticated heuristics.

Metrics

Few critical metrics were proposed, again, in a simplified fashion, definitely not exhaustive.

- Average Test Case Duration - How much does an interaction with a chatbot take?
- Tool Selection Accuracy - Is the bot selecting the right tools?
- Parameter Extraction Accuracy - Is the bot setting parameters for the tools correctly?
- Correct Tool Invocation Rate - The Right tool and right parameter selection.
- Tool Call Success Rate (Technical, Excl. Expected Failures) - Are the tools working correctly?
- Average Tool Execution Latency (Successful Calls) - How much does it take to call a tool?
- Cancellation Flow Compliance - Is the bot compliant with the cancellation policy?

Skipped metrics

While several key metrics were tracked to evaluate the assistant's performance (e.g., tool selection accuracy, parameter extraction, cancellation flow compliance), a more comprehensive evaluation could include additional metrics that were skipped in this iteration:

- **LLM Token Usage / Cost:** Tracking the number of prompt and completion tokens used per interaction. This metric would directly measure the operational cost associated with the language model, relevant to the non-functional requirement of keeping the solution cost-effective.
- **Response Quality Score:** A quantitative assessment of the assistant's natural language responses, evaluating factors like clarity, helpfulness, politeness, and accuracy in explaining policies. This goes beyond technical correctness to measure the quality of the user interaction.
- **Concurrency Success Rate:** Although concurrency tests were out of scope, if they were implemented, this metric would measure the system's ability to handle simultaneous user requests successfully, indicating its robustness under load.

Summary of evaluation

The evaluation involved running 11 distinct test cases covering the core functionalities: tracking existing and non-existent orders, adding orders, listing orders, checking cancellation eligibility (for eligible, ineligible due policy, non-existent, boundary 10/11 day cases, and already cancelled orders), and handling simulated network errors during tool execution.

The analysis of the execution logs against the ground truth yielded very positive results:

- **Tool Selection & Parameter Extraction:** The assistant demonstrated 100% accuracy in selecting the correct tool for the user's intent and extracting the necessary parameters (like order_id or item_name) for all 11 test cases where a tool call was attempted.
- **Correct Tool Invocation:** Consequently, the correct tool was invoked with the correct parameters in 100% of the attempts.
- **Technical Tool Success Rate:** Excluding one test case designed to simulate a network failure (which failed as expected), the underlying tool calls (API interactions) were technically successful 100% of the time (10 out of 10 relevant calls).
- **Cancellation Policy Compliance:** In all 6 test cases designed to check the cancellation flow, the assistant correctly determined eligibility based on the 10-day policy and appropriately either requested user confirmation (for eligible orders) or informed the user of ineligibility (for orders too old, not found, or already cancelled), achieving 100% compliance.
- **Performance:** The average latency for successful tool executions (API calls) was very low at approximately 2.03 ms (though this reflects the speed of the mock API). The average duration for the agent to process a query and respond (including LLM calls and tool execution) across the test cases was about 2.55 seconds.

Overall, the evaluation indicates that the chosen design ("LLM with functions and human approval") performs reliably and accurately according to the defined requirements and test scenarios. It correctly handles core order management tasks, adheres strictly to the cancellation policy, and manages basic error conditions appropriately. While more comprehensive testing (covering skipped areas like concurrency and response quality scoring) could provide further insights, the current results validate the effectiveness of the implemented solution.

Conclusion/summary

This report detailed the development and evaluation of an Order Assistant designed to meet specific functional requirements for order management and non-functional requirements for safety and cost-effectiveness. The implemented "LLM with functions and human approval" design proved effective.

Evaluation highlights include:

- **Accuracy:** 100% accuracy was achieved in tool selection, parameter extraction, and correct tool invocation across the test cases.

- **Policy Compliance:** The assistant demonstrated 100% compliance with the 10-day cancellation policy, correctly identifying eligibility and managing the user confirmation flow as designed.
- **Error Handling:** Basic network errors during tool execution were handled gracefully.

While the evaluation was simplified (omitting areas like full concurrency testing and response quality scoring), the results strongly validate that the chosen solution meets the core requirements. The assistant reliably performs order management tasks and incorporates the necessary safety layer for cancellations, confirming the suitability of the selected design for this use case.