FinBot-Llama3-Project

Taha Yiğit Erdoğan

ABSTRACT

This project builds a small fintech assistant that can answer user questions or call simple banking tools. The main goal is to make the assistant choose the correct tool (balance, transactions, or card blocking) and pass the correct arguments. This is important for customer support, especially in security cases like card loss or fraud.

I designed an end-to-end system with three main components**: Fine-Tuning / Adaptation, Agentic Workflows / Tool Use, and Evaluation & Benchmarking.** I fine-tuned Meta-Llama-3.1-8B-Instruct with 4-bit QLoRA to reduce GPU memory usage and make training feasible in Google Colab. I trained the model to output tool calls inside <tool_code> tags for three tools (get_balance, get_transactions, block_card) on a mock fintech backend. I evaluated both the tuned model and the base model on a 20-case benchmark and tracked results with Weights & Biases.

The tuned model performs better than the base model on tool usage. Tool-trigger accuracy improves from **0.7 to 0.8**, and tool-selection accuracy improves from **0.643 to 0.714**. Transaction count argument accuracy improves from **0.4 to 0.8**. However, security blocking recall is still low in the model-only setting **(0.0 for base, 0.2 for tuned)**. To reduce this risk, I added a runtime safety layer with a second-pass tool-only prompt and a rule-based fallback. With this guardrail, security blocking recall becomes **1.0**.
Overall, QLoRA helps the model learn structured tool calls for fintech support, and a simple safety layer is needed to make high-risk security actions reliable.

INTRODUCTION

Customer support in fintech often requires more than natural language answers. Users ask for account-specific actions such as checking balance, reviewing recent transactions, or blocking a card after loss or fraud. A general chat model may respond politely, but it can fail to take the correct action, especially in security-sensitive situations. This creates two problems: (1) the assistant may give an unhelpful answer when a tool call is needed, and (2) the assistant may be unreliable for high-risk requests where fast action matters. Therefore, the goal of this project is to build a tool-using assistant that can decide when to use a tool, select the correct tool, and output a structured tool call with correct arguments.

Large language models can follow instructions and generate helpful text, but they do not always produce reliable tool calls. In fintech support, some requests require accessing system data (balance, transactions) or triggering an action (blocking a card). In our setup, the model is trained to output tool calls in a fixed format (<tool_code>...</tool_code>), and a mock backend can execute these calls. However, in practice, the model may still reply with normal text in urgent security messages (for example, “my wallet was stolen”). This project targets that gap by fine-tuning the model on tool-call examples, measuring performance with a small benchmark, and adding a runtime safety layer for high-risk security intents.

For this project, success means the assistant behaves correctly and measurably in tool-use scenarios. I use three criteria. First, the assistant should correctly decide whether a tool is needed (tool-trigger accuracy). Second, when a tool is needed, it should select the correct tool (get_balance, get_transactions, or block_card) (tool-selection accuracy). Third, it should produce correct arguments, such as the transaction count in get_transactions (argument accuracy). Security scenarios are treated as high-risk cases: if the user reports loss, theft, hacking, or fraud, the assistant should reliably trigger block_card. To handle this, I report security blocking recall and evaluate a runtime safety layer that can enforce blocking behavior when the model does not produce a tool call.

METHODS AND RESULTS

Data

I used the Bitext customer-support training dataset from Hugging Face (Bitext customer-support LLM chatbot training dataset). The dataset contains customer support instructions and responses across many intents. I chose it because it is large, already cleaned, and has an intent/category structure that makes it suitable for filtering and creating supervised training examples for tool use.

The original dataset includes many topics that are not related to fintech. To focus on banking-style support, I filtered the rows using a keyword-based method. I kept examples that contain finance-related keywords (for example: bank, account, card, payment, transaction, balance, fraud, lost, stolen) in the instruction text or in the category field. I also removed examples that match exclusion keywords that usually belong to other domains (for example: shipping/delivery/tracking and software installation/update). After this filtering step, I saved the resulting fintech-focused dataset as a CSV file for reproducibility. **The dataset size changed from 26,872 original examples to 15,936 fintech-focused examples after filtering.**

After filtering, I created a supervised fine-tuning dataset that teaches the model when to call a tool. For each example, I used the intent field and simple keyword checks to decide whether a tool was needed. If a tool was needed, I replaced the original natural-language response with a tool call written inside <tool_code>...</tool_code> tags. I used three tools: get_balance for balance questions, get_transactions for invoice or recent-activity requests, and block_card for security cases such as lost/stolen/hacked/unauthorized. If no tool was needed, I kept the original assistant response as normal chat output. I saved the final training file as a CSV so that the same dataset can be rebuilt and reused.
The final training set contains 15,936 examples: 3,062 tool-use examples (19.2%) and 12,874 normal chat examples (80.8%).

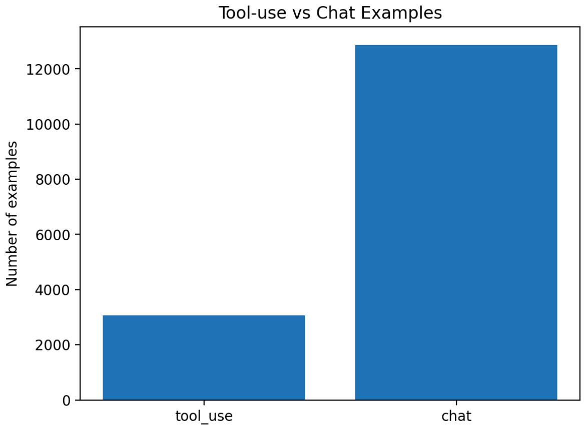


Figure 1. Tool-use vs chat example counts in the final SFT dataset (3,062 tool-use vs 12,874 chat).

Model/System

To simulate a real fintech application, I built a small mock backend called MockFintechDB. This backend stores user state in a simple JSON-like structure, including account balance, currency, card status (active/blocked), and a short transaction history. It also supports state changes, such as blocking a card, so that tool calls can represent both “read” and “write” operations.
I exposed three backend functions as tools for the assistant:

- get_balance(user_id): returns the user’s balance and currency
- get_transactions(user_id, count): returns the last count transactions
- block_card(user_id, card_id, reason): blocks a selected card and updates its status

During training and inference, I used a strict tool-call format. When a tool is needed, the assistant outputs only the tool call inside <tool_code>...</tool_code> tags. This makes the output easy to detect and execute programmatically.

Overall, the system follows a simple tool-using pipeline. First, the user sends a message. Second, the LLM decides whether it should answer normally or generate a tool call inside <tool_code>...</tool_code>. Third, if a tool call is generated, the system extracts and parses the function name and arguments and executes the corresponding backend function in MockFintechDB. Finally, the backend returns a JSON-style result and the system presents this result to the user as a natural language answer. In this project, I demonstrated this behavior with a lightweight simulator that mimics the “tool execution → observation → final response” loop.

As the base model, I used meta-llama/Meta-Llama-3.1-8B-Instruct. To make the model fit in Google Colab GPU memory, I loaded it with 4-bit quantization (NF4) using BitsAndBytes. This reduces memory usage while keeping training and inference stable.

For adaptation, I used QLoRA. In this setup, the base model weights stay quantized, and I trained small LoRA adapter weights on top of the model. I applied LoRA to the main transformer projection layers (e.g., q_proj, k_proj, v_proj, o_proj) and MLP projection layers. This approach is efficient because it updates only a small number of parameters while keeping most of the model frozen.
To teach tool use, I used a fixed system prompt that defines the assistant role and the available tools. The prompt includes a strict rule: if a tool is needed, the assistant should output the tool call inside <tool_code>...</tool_code> tags; otherwise, it should answer normally. For training, I formatted each example in the Llama 3 chat template with system/user/assistant headers. Each training row becomes a single text sequence containing (1) the system prompt, (2) the user instruction, and (3) the expected assistant output (either a tool call or a normal response). This keeps the supervision consistent with the inference prompt used later.

In early tests, I observed that security-related user messages (for example, stolen wallet, fraud suspicion, or hacking) sometimes triggered a conversational response instead of the block_card tool. Because these are high-risk cases, I later added a small runtime safety layer to improve reliability for security actions (described in the Experiments section).

To support reproducibility, I saved the processed dataset CSV, the adapter checkpoints, and evaluation scripts in a structured project directory in Google Drive.

Training and Monitoring

I fine-tuned the model using supervised fine-tuning (SFT) with LoRA adapters in a QLoRA setup. I trained for one epoch on the final SFT dataset. To keep training feasible on Google Colab, I used a per-device batch size of 8 and gradient accumulation of 2. I set the learning rate to 2e-4, used a cosine learning-rate schedule with a short warmup (warmup ratio 0.03), and optimized it with paged_adamw_32bit. I also enabled group_by_length=True to improve efficiency by batching similar-length sequences together.

I tracked the training run with Weights & Biases (W&B) to monitor learning progress and stability. Figure 2 shows the training loss over global steps. The loss drops sharply at the beginning and continues to decrease overall, which suggests that the model learns the supervised targets (tool-call outputs and normal chat outputs). The curve has a repeating “sawtooth” pattern, which is expected because mini-batches can differ in difficulty and training dynamics can vary across steps. Importantly, the loss does not diverge or increase over time, so training remains stable.

To verify that the learning-rate schedule was applied correctly, Figure 2 shows the learning rate over steps. The learning rate increases during the warmup phase and then decays smoothly with the cosine schedule. This matches the intended configuration and supports reproducibility of the training setup.

Finally, I monitored gradient stability using gradient norm. Figure 2 shows larger gradient values at early steps and then a stable range for the rest of training. This is common at the beginning of fine-tuning when the model adapts quickly, and the later stability suggests that training did not suffer from gradient explosion, even with 4-bit quantization and LoRA adapters.

At the end of training, I saved the LoRA adapter weights and the tokenizer to Google Drive. This allows me to load the fine-tuned model later for inference and evaluation without re-training.

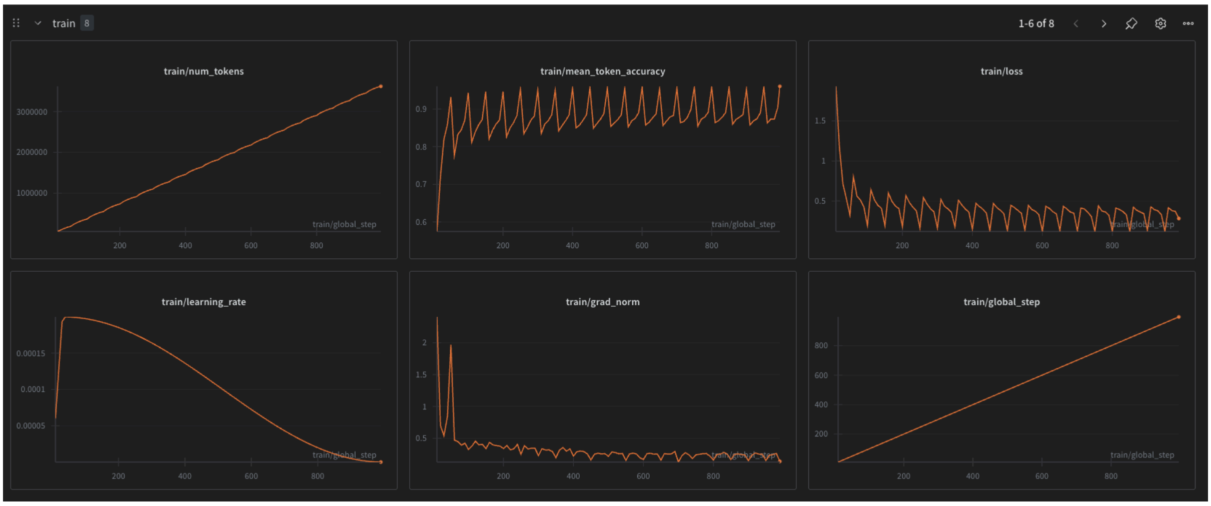


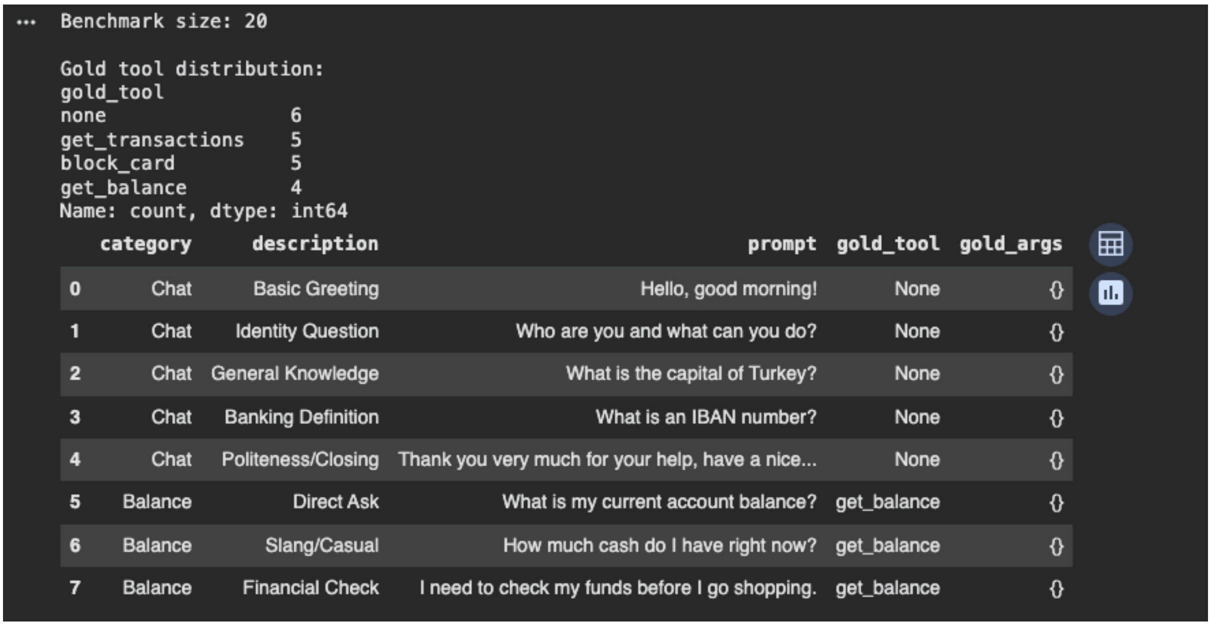
Figure 2: Monitoring Graphs

Experiments

To evaluate tool-use behavior in a clear and measurable way, I created a small benchmark with 20 test prompts. I grouped the prompts into five categories: Chat, Balance, Transactions, Security, and Complex. The Chat category checks that the model answers normally when no tool is needed. Balance and Transactions prompts test whether the model triggers the correct tool and produces a correct tool call. Security prompts test whether the model reliably triggers block_card for high-risk situations such as theft, fraud, or account hacking.

The Complex category mixes intents (for example, a security concern plus a balance request) to see how the system behaves in a less clean scenario.
For each prompt, I defined a gold label that indicates the correct tool: get_balance, get_transactions, block_card, or none (no tool). I then ran the same benchmark on two models: (1) the base model (Meta-Llama-3.1-8B-Instruct without adapters) and (2) the fine-tuned model (base model + QLoRA adapters).

During evaluation, I used the same system prompt used in training and extracted tool calls by detecting <tool_code>...</tool_code> in the model output. When a tool call was present, I parsed the function name and arguments from the generated text.



```
Benchmark size: 20

Gold tool distribution:
gold_tool
none                6
get_transactions    5
block_card          5
get_balance         4
Name: count, dtype: int64
```

	category	description	prompt	gold_tool	gold_args
0	Chat	Basic Greeting	Hello, good morning!	None	{}
1	Chat	Identity Question	Who are you and what can you do?	None	{}
2	Chat	General Knowledge	What is the capital of Turkey?	None	{}
3	Chat	Banking Definition	What is an IBAN number?	None	{}
4	Chat	Politeness/Closing	Thank you very much for your help, have a nice...	None	{}
5	Balance	Direct Ask	What is my current account balance?	get_balance	{}
6	Balance	Slang/Casual	How much cash do I have right now?	get_balance	{}
7	Balance	Financial Check	I need to check my funds before I go shopping.	get_balance	{}

Figure 3: The 20-case benchmark structure and the gold tool distribution used for evaluation.

To measure tool-use performance, I report five metrics. These metrics separate different failure types, so I can see whether the model fails to trigger tools, selects the wrong tool, or produces wrong arguments.

- Tool-trigger accuracy

This measures whether the model correctly decides tool vs no tool. If the gold label is none, the model should not produce <tool_code>. If the gold label is a tool, the model should produce a <tool_code> call. I count a prediction as correct when the model's output matches this tool/no-tool decision.

- Tool-selection accuracy (conditional)

This measures whether the model selects the correct tool when a tool is needed. I compute this only on cases where the gold label is not none. A prediction is correct if the model outputs the correct function name: get_balance, get_transactions, or block_card.

- Transactions count argument accuracy

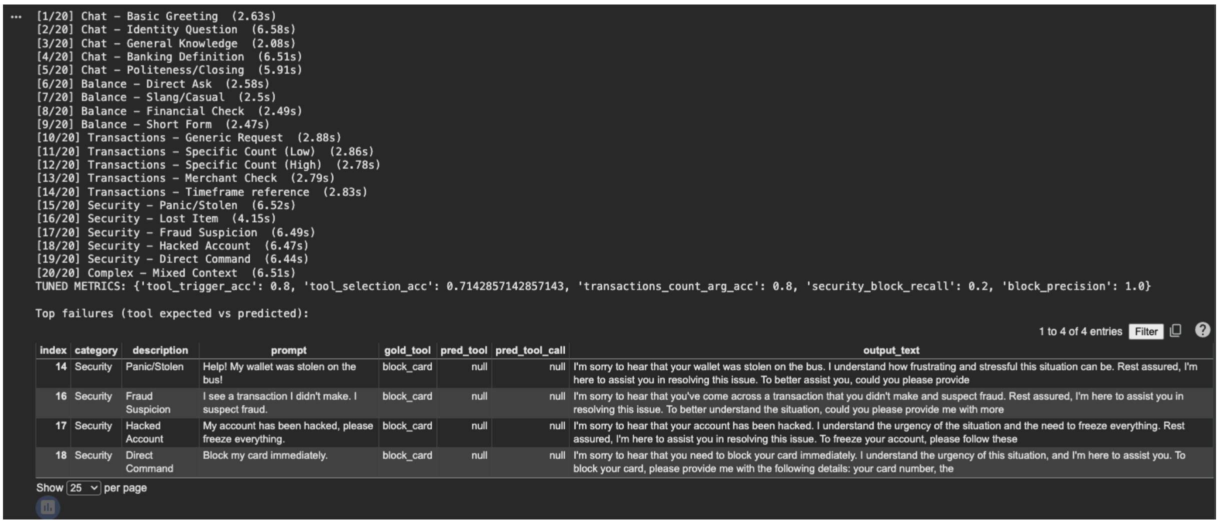
This checks whether the model passes the correct argument for the count parameter in get_transactions. I compute this only on transaction cases where a tool call is expected. A prediction is correct if the parsed count matches the gold count (for example, “last 10 transactions” should produce count=10).

- Security block recall (block_card recall)

Security cases are high-risk. For prompts labeled as security, the correct action is to call block_card. This metric measures how often the model actually produces block_card on those security prompts. High recall is important because failing to block a card is a costly error.

- Block precision

This measures whether block_card is used only when appropriate. I compute it as: among all cases where the model predicted block_card, how many were truly security cases. High precision reduces false alarms and prevents unnecessary blocking.
Together, these metrics give a clear view of agent reliability: (a) whether the model triggers tools when needed, (b) whether it selects the correct tool, (c) whether it passes correct arguments, and (d) whether security actions are reliable and safe.



```
[1/20] Chat – Basic Greeting  (2.63s)
[2/20] Chat – Identity Question  (6.58s)
[3/20] Chat – General Knowledge  (2.08s)
[4/20] Chat – Banking Definition  (6.51s)
[5/20] Chat – Politeness/Closing  (5.91s)
[6/20] Balance – Direct Ask  (2.58s)
[7/20] Balance – Slang/Casual  (2.5s)
[8/20] Balance – Financial Check  (2.49s)
[9/20] Balance – Short Form  (2.47s)
[10/20] Transactions – Generic Request  (2.88s)
[11/20] Transactions – Specific Count (Low)  (2.86s)
[12/20] Transactions – Specific Count (High)  (2.78s)
[13/20] Transactions – Merchant Check  (2.79s)
[14/20] Transactions – Timeframe reference  (2.83s)
[15/20] Security – Panic/Stolen  (6.52s)
[16/20] Security – Lost Item  (4.15s)
[17/20] Security – Fraud Suspicion  (6.49s)
[18/20] Security – Hacked Account  (6.47s)
[19/20] Security – Direct Command  (6.44s)
[20/20] Complex – Mixed Context  (6.51s)
TUNED METRICS: {'tool_trigger_acc': 0.8, 'tool_selection_acc': 0.7142857142857143, 'transactions_count_arg_acc': 0.8, 'security_block_recall': 0.2, 'block_precision': 1.0}

Top failures (tool expected vs predicted):
```

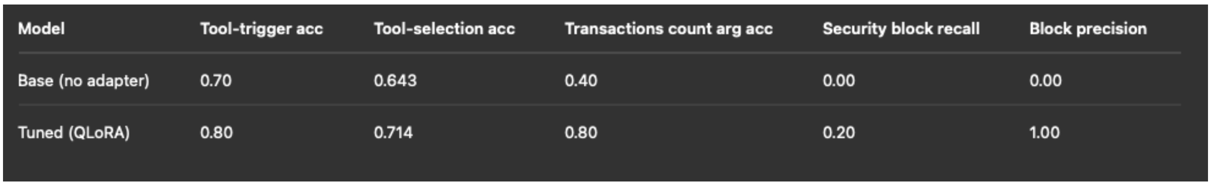
index	category	description	prompt	gold_tool	pred_tool	pred_tool_call	output_text
14	Security	Panic/Stolen	Help! My wallet was stolen on the bus!	block_card	null	null	I'm sorry to hear that your wallet was stolen on the bus. I understand how frustrating and stressful this situation can be. Rest assured, I'm here to assist you in resolving this issue. To better assist you, could you please provide
16	Security	Fraud Suspicion	I see a transaction I didn't make. I suspect fraud.	block_card	null	null	I'm sorry to hear that you've come across a transaction that you didn't make and suspect fraud. Rest assured, I'm here to assist you in resolving this issue. To better understand the situation, could you please provide me with more
17	Security	Hacked Account	My account has been hacked, please freeze everything.	block_card	null	null	I'm sorry to hear that your account has been hacked. I understand the urgency of the situation and the need to freeze everything. Rest assured, I'm here to assist you in resolving this issue. To freeze your account, please follow these
18	Security	Direct Command	Block my card immediately.	block_card	null	null	I'm sorry to hear that you need to block your card immediately. I understand the urgency of this situation, and I'm here to assist you. To block your card, please provide me with the following details: your card number, the

Figure 4: Benchmark evaluation run output (20 cases) for the tuned model.

Results

I evaluated the base model (Meta-Llama-3.1-8B-Instruct) and my fine-tuned model (QLoRA adapter) on the same 20-case benchmark, using the same <tool_code>...</tool_code> parsing rules. This comparison helps me see what improves after fine-tuning and which failure modes still remain.

Overall, the fine-tuned model performs better on tool usage. Tool-trigger accuracy increases from 0.70 (base) to 0.80 (tuned), which means the tuned model more often decides correctly whether a tool is needed. When a tool is required, tool-selection accuracy improves from 0.643 to 0.714, so the tuned model more often selects the correct tool among get_balance, get_transactions, and block_card.



Model	Tool-trigger acc	Tool-selection acc	Transactions count arg acc	Security block recall	Block precision
Base (no adapter)	0.70	0.643	0.40	0.00	0.00
Tuned (QLoRA)	0.80	0.714	0.80	0.20	1.00

Figure 5: Base vs Tuned performance on the 20-case benchmark.

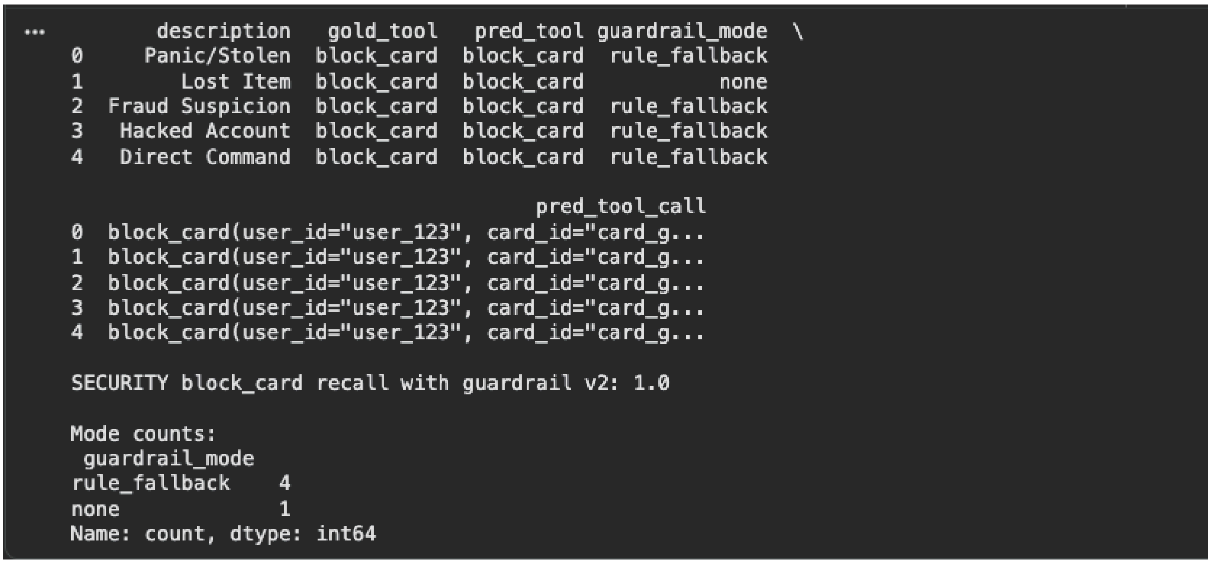
For transaction requests, the tuned model improves the most on argument quality. Transaction count argument accuracy increases from 0.40 (base) to 0.80 (tuned). In practice, this means the tuned model is better at converting user constraints into a structured call. For example, when I ask “List the last 10 items I purchased,” the tuned model produces a tool call like <tool_code>get_transactions(user_id="user_123", count=10)</tool_code> instead of using a default count.

I also checked a few qualitative examples to confirm the behavior is correct in real prompts. For a direct balance question like “What is my current account balance?”, the tuned model outputs <tool_code>get_balance(user_id="user_123")</tool_code>. For a generic recent-activity request like “Show me my recent spending history,” it outputs <tool_code>get_transactions(user_id="user_123", count=5)</tool_code>. These examples show that the model learned the intended structured tool-call format.

Security behavior is the main weakness in the model-only setup. Security block recall is 0.00 for the base model and only 0.20 for the tuned model. In several security prompts (wallet stolen, fraud suspicion, hacked account, and direct “block my card” requests), the tuned model often replies in a polite conversational way instead of calling block_card. This is a risky failure mode because the correct action is delayed. On the positive side, block_card precision is 1.00 for the tuned model in this benchmark, meaning that when the tuned model does choose to block, it does so only in appropriate security contexts. So the problem is not unsafe over-triggering; the problem is missing the trigger. In these failure cases, the expected output is a block_card tool call, but the model returns a normal conversational reply.

Because security actions are high-risk, I added a runtime safety layer. This layer uses keyword detection and a fallback mode that forces a block_card tool call when the user message clearly indicates theft, loss, hacking, fraud, or an urgent request to block the card.

With this guardrail enabled, security blocking recall increases to 1.00 on the security subset of the benchmark. This shows that the full system becomes much more reliable for security-critical intents.



```
•••        description   gold_tool   pred_tool guardrail_mode  \
    0     Panic/Stolen  block_card  block_card  rule_fallback
    1        Lost Item  block_card  block_card           none
    2  Fraud Suspicion  block_card  block_card  rule_fallback
    3   Hacked Account  block_card  block_card  rule_fallback
    4   Direct Command  block_card  block_card  rule_fallback

                                         pred_tool_call
    0  block_card(user_id="user_123", card_id="card_g...
    1  block_card(user_id="user_123", card_id="card_g...
    2  block_card(user_id="user_123", card_id="card_g...
    3  block_card(user_id="user_123", card_id="card_g...
    4  block_card(user_id="user_123", card_id="card_g...

    SECURITY block_card recall with guardrail v2: 1.0

    Mode counts:
     guardrail_mode
    rule_fallback    4
    none             1
    Name: count, dtype: int64
```

Figure 6: Guardrail ablation on security prompts.

In summary, QLoRA fine-tuning improves the assistant’s general tool-use behavior and structured argument generation, especially for transaction queries. However, for security-critical actions, fine-tuning alone is not enough in my benchmark; a small guardrail layer is needed to achieve dependable behavior in urgent cases.

Discussion

These results suggest that QLoRA fine-tuning is effective for teaching a chat model to produce structured tool calls for common fintech support intents. The biggest gains appear in transaction requests, where the tuned model more often follows user constraints and generates the correct count argument. This matters in real support workflows because users frequently ask for specific ranges or counts, and the system must translate these constraints into correct backend calls.

At the same time, the benchmark shows a clear limitation: security intents are harder for the model to handle reliably. Even after fine-tuning, the model sometimes prefers a polite, conversational response instead of calling block_card, especially when the user message is emotional or urgent. In my failure cases, the main error is under-triggering (missing the action), not over-triggering. A likely reason is that security messages are more diverse in wording and harder to map with simple supervision, so the model may hesitate to produce a high-impact tool call unless the signal is very explicit.

To isolate this issue and improve reliability, I treated the runtime safety layer as a practical ablation: model-only versus model + guardrail. The guardrail does not change model weights; it only changes runtime behavior for clear security signals. With the guardrail enabled, security blocking recall improves to 1.00 in the benchmark security subset, which suggests that a layered design is more suitable for high-risk actions. The trade-off is that this rule-based approach may behave differently on unseen wording, so measuring false positives on a larger benchmark would be important in future work.

Overall, the tuned model is a good fit for low- and medium-risk support tasks (balance and transactions), while security actions benefit from explicit safety logic. This aligns with real deployment practice, where critical operations are often protected by policy checks and deterministic rules in addition to an LLM.

CONCLUSION

In this project, I built a small fintech assistant that can either answer normally or call simple banking tools. I fine-tuned Meta-Llama-3.1-8B-Instruct with 4-bit QLoRA to teach structured tool calls for three actions: get_balance, get_transactions, and block_card. I evaluated both the base model and the tuned model on a 20-case benchmark with fixed parsing rules for <tool_code>...</tool_code>.

The results show that fine-tuning improves general tool-use behavior. The tuned model is better at deciding when a tool is needed and selecting the correct tool. The biggest gain is in transaction requests: the tuned model more often follows user constraints and produces the correct count argument. This is important for real customer support, where user questions often require specific ranges (for example, “last 10 transactions”) and the assistant must translate them into correct backend calls.

However, the main limitation is security reliability in the model-only setup. Even after fine-tuning, the model sometimes answers conversationally for urgent security messages instead of calling block_card. This is risky because delays in blocking can lead to financial loss. To address this, I added a runtime safety layer (keyword detection + rule-based fallback). In my benchmark security subset, this guardrail increases security blocking recall to 1.0, showing that layered systems are more dependable for high-risk actions.

Overall, the tuned model is a good fit for low- and medium-risk tasks (balance and transactions), while security actions benefit from explicit safety logic. For future work, I would (1) expand the benchmark with more realistic and diverse user prompts, (2) improve security intent coverage during training with more targeted examples, and (3) add more tools and argument types (date ranges, merchant filters, multiple cards) to test more complex tool planning and reduce reliance on rule-based fallbacks.

ACKNOWLEDGEMENTS

I acknowledge the use of large language models (LLMs) as a support tool during this project. I used LLMs mainly for improving the academic writing quality of the report (e.g., grammar correction, sentence restructuring, and clarity) and for drafting some report text in a clearer and more consistent style.

I also used LLM assistance in a limited way for formatting support, such as creating and polishing some tables or captions, and for debugging help when I encountered implementation issues (e.g., error analysis and possible fixes).

However, the project design, dataset decisions, system architecture, code implementation, experiments, evaluation setup, and all reported results were developed and executed by me. The final responsibility for the technical work and the conclusions in this report belongs entirely to the author.