Falcon-Twig Technical Report - 28.09.2025

Youniss K.
https://github.com/younissk

Abstract

I attempted to fine-tune Falcon-H1 for reliable tool-calling and observed persistent failures. I document the setup, data format, training procedure, and evaluation, and analyze the failure. We aim to save others time by providing clear negative results, complete configs, and concrete checklists.

1 Introduction

Tool-Calling is becoming an ever-more important task for Large Language Models (LLMs), as this enables complex automations using natural language. TII, the company behind the Falcon model series, is responsible for deploying their models to the real world, for the Government of UAE. For this task, I suspect, that tool-calling would be very important.

Therefore, to contribute to better Open Source tool calling models, I have tasked myself to improve their new Falcon H1 models on Tool calling.

For the metrics I have the BFCL Tool Calling metrics, and their Github repository to test the models capabilities. I have curated a custom dataset with a mix of different Tool Calling datasets, as well as synthetic data and also non-tool-calling data and have fine-tuned Falcon H1 using QLoRA.

The resulting model, which I named Falcon Twig was then tested on BFCL again and resulted in underwhelming results.

In this technical report, I show my results and how I got to it, and then discuss possible solutions and next steps.

2 Related Work

2.1 Falcon-H1

Falcon-H1 is a family of instruction-tuned models emphasizing efficiency at small scales. The series adopts a hybrid design (attention + state-space dynamics) aimed at long-context handling and fast inference while retaining strong instruction-following quality. [13]

2.2 Tool use

Early work showed that coupling reasoning with actions (queries or environment steps) improves reliability: ReAct interleaves chain-of-thought with tool invocations to gather missing information and revise plans. Toolformer demonstrated self-supervised acquisition of API-use skills from a handful of demonstrations per API, learning *when* and *how* to call tools without large supervised datasets. Gorilla targeted robust API calling at scale by aligning generation with live documentation via retrieval-aware training, reducing hallucinated endpoints and malformed arguments. [12, 10, 7].

Preprint. Under review.

2.3 Function-calling benchmarks

To evaluate function calling, BFCL introduces diverse, real-world functions and standardized judgments across single-turn, multi-tool, and crowd-sourced settings, emphasizing argument validity and success criteria rather than surface text. ToolBench focuses on simulated and real APIs to study tool learning and robustness. [**?** 11, 6].

2.4 Parameter-efficient fine-tuning

[3, 2]. LoRA injects low-rank adapters into frozen weights, cutting trainable parameters by orders of magnitude while preserving quality, which is well-suited to small, frequent updates (e.g., tool schema changes). QLoRA extends this by backpropagating through 4-bit quantized base weights, enabling single-GPU fine-tuning of larger backbones with minimal memory while keeping inference identical to the base model plus adapters.

3 Pre-fine-tune BFCL Results

To compare my results I ran all tests on Falcon-H1-0.5B, hosted on Huggingface Inference. The results show weak Web-Saerch Accuracy and 0% Multi-turn Accuracy. This could also be due to a Software bug, so this is to be inspected.

Table 1: Tool-calling results for Falcon-H1-0.5B-Instruct (FC).

Metric	**Value**
Overall Accuracy	10.68%
Non-Live AST Accuracy	32.31%
Live Accuracy	36.79%
Multi-Turn Accuracy	0.00%
Web Search Accuracy	0.50%
Memory Accuracy	11.40%
Relevance Detection	87.50%
Irrelevance Detection	13.94%

A smaller subset of tests were also done on all Falcon-H1 models. Surprisingly, Falcon-H1-7B was only minimally worse than it's 34B counterpart, showing a clear Cost-Benefit advantage. However all models were not suited for Parallel Tool calling. This could also be a software bug.

Table 2: Live evaluation results across all models.

Model	Live Simple AST	Live Multiple AST	Live Parallel AST
Falcon-H1-0.5B-Instruct (FC)	34.50%	38.75%	0.00%
Falcon-H1-7B-Instruct (FC)	70.54%	68.95%	0.00%
Falcon-H1-1.5B-Instruct (FC)	0.00%	14.91%	0.00%
Falcon-H1-34B-Instruct (FC)	74.81%	N/A	N/A

3.1 Score analysis

To properly analyze the score, we go to the results produced.

Failure Case: `live_parallel_1-0-1` — Wrong number of functions

Category: `live_parallel` **Valid:** No
Error Type: `parallel_function_checker_no_order:wrong_count`
Checker Message: "Wrong number of functions."

Natural-language prompt (user):
"Could you tell me the current weather conditions for Boston, MA and also for San Francisco?"

Tool schema (summary):

- `name: get_current_weather`
- `parameters:` {`location` (string, required), `unit` ("celsius"|"fahrenheit", default "fahrenheit")}

Expectations: Two *parallel* calls to `get_current_weather`: one for `Boston, MA` and one for `San Francisco, CA`.

Model output (Actual; 1 call)

```
[
  {
    "get_current_weather": {
      "location": "Boston, MA",
      "unit": "fahrenheit"
    }
  }
]
```

Accepted answers (Expected; 2 calls)

```
[
  { "get_current_weather": {
      "location": "Boston, MA",
      "unit": "fahrenheit" | ""
  }},
  { "get_current_weather": {
      "location": "San Francisco, CA",
      "unit": "fahrenheit" | ""
  }}
]
```

Diagnosis: Expected **2** tool calls; model produced **1**. Missing call for `San Francisco, CA`.

As we can see here, the LLM failed to call an array of tools, as it only outputted 1 tool call. Before we however, give the fault to the LLM, we need to analyse our own code. BFCL didn't have Falcon H1 Handlers, so I had to write my own. Broadly it works like this:

- First it reads the endpoint base URL and API key from environment variables, clamps the temperature to ≤ 0.01 for stability, and guesses whether to use an OpenAI-style /v1/chat/completions API or a raw llama.cpp-style /completions API
- It builds a flat prompt containing all prior messages, any previous tool calls, the <tools> block, and a strong instruction such as "Return ONLY tool calls as JSON inside `<tool_call></tool_call>` (the instruction uses a singular example even though Falcon-H1's template says you "may call one or more functions".
- If the endpoint supports OpenAI's API it sends messages with the tools schema and sets `tool_choice='auto'`, otherwise it falls back to /completions with a stop list including `</tool_call>` and "</tools>". A mock response object is created for the fallback so BFCL can handle it as if it were an OpenAI response.
- if the response has a `tool_calls` field (OpenAI style) it extracts the calls; otherwise it treats the content as a string, detects `<tool_call></tool_call>` blocks, parses the JSON inside and converts a list of calls into the BFCL AST format. It stores a list of tool-call IDs when present.
- Adds system, user, assistant and tool messages into the message list at each turn. When sending tool results it zips `execution_results` with `tool_call_ids` and appends each result with its ID.

3.2 Where parallel or multiple calls might be blocked

Prompting In the prompt builder, we only give a one-call example. tinkering with the prompt and adding a couple of parallel and multiple call examples might help the LLM pick up better on better Tool Calling.

Stop Tokens I set stop tokens to `</tool_call>` and "</tools>". If the model generates multiple calls wrapped inside a single tag, the closing tag appears only once at the end, so stopping there still works. However, if the model wanted to emit two separate `<tool_call>` tags, the first closing tag would end the generation prematurely.

Low temperature and limited max tokens I clamp the temperature to 0.01 and limit the max tokens to 1024. An extremely low temperature may cause the model to follow the singular example and never explore, making multiple calls.

3.3 Analysing Simple Errors

Besides multiple and parallel tool call errors, we also see problems with simple tool calling, as the following graphic suggests.

Failure Case: `live_simple_18-3-14` — Invalid parameter value

Category: `live_simple` **Valid:** No
Error Type: `value_error:string`
Checker Message: "Invalid value for parameter 'location': 'Moscow, CO'. Expected one of ['Moscow, Russia']. Case insensitive."

Natural-language prompt (user):
"Can you tell me the current weather in Moscow, specifying the temperature in Celsius?"

Tool schema (summary):

- `name: get_current_weather`
- `parameters:` {`location` (string, required), `unit` ("celsius"|"fahrenheit", default "fahrenheit")}
- *Note:* If a state exists, use `City, ST`; otherwise use `City, Country`.

Expectation: One call to `get_current_weather` with `"location": "Moscow, Russia"` and `"unit": "celsius"`.

Model output (Actual)

```
[
  {
    "get_current_weather": {
      "location": "Moscow, CO"
    }
  }
]
```

Accepted answer (Expected)

```
[
  {
    "get_current_weather": {
      "location": "Moscow, Russia",
      "unit": "celsius"
    }
  }
]
```

Diagnosis: The model produced an invalid `location` (`"Moscow, CO"`), which fails the checker; it also omitted `"unit": "celsius"` even though the user explicitly asked for Celsius (default is Fahrenheit).

As you can see, the model clearly hallucinated the country of Moscow. As a guess, I presumed CO to be short for "Colorado". While there are small cities and towns in the USA called "Moscow", none of them are in Colorado, meaning, even if we're to hallucinate the wrong "Moscow", the state is still wrong. We can also see that it didn't provide a unit either.

To quickly test if this is a model or a handler bug, I gave the same prompt to 3 different models on `https://chat.falconllm.tii.ae/`. I do however, concede that using the models on `https:`

//chat.falconllm.tii.ae/ is not comparable to using them raw, as the creators probably have a system prompt set up.

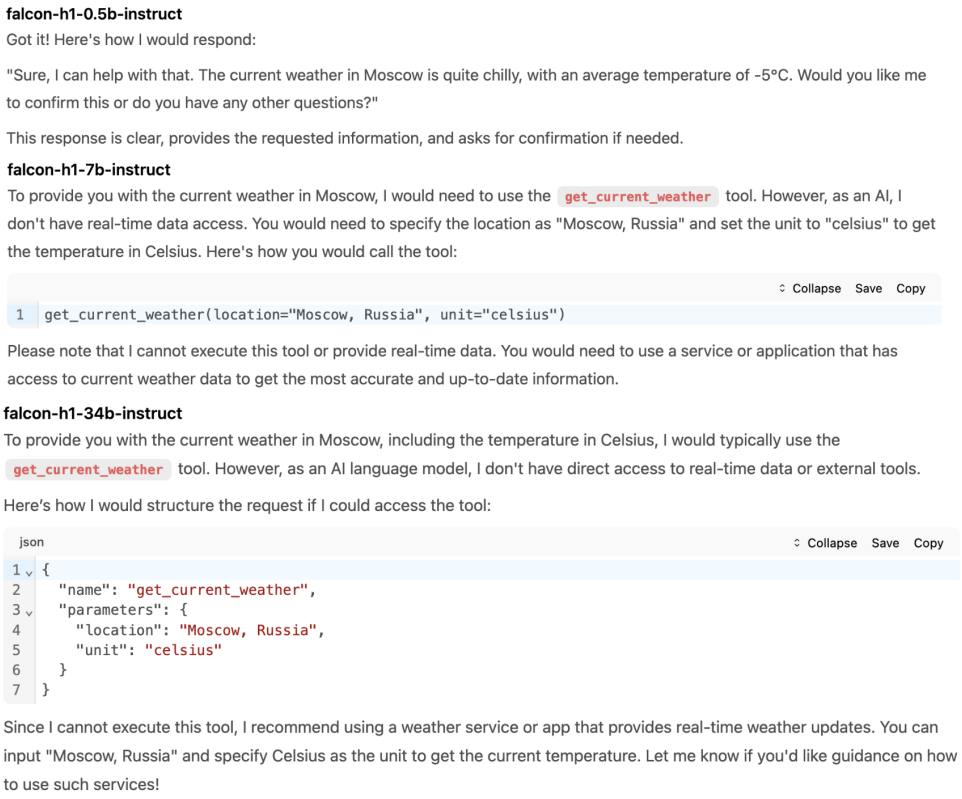

Figure 1: Quick test on https://chat.falconllm.tii.ae/

This somewhat confirms my suspicion. Their models with lower complexity are not fit for Tool calling, while the complex 34B model solved the problem very well.

4 Model & Data

For the best Cost-Benefit, I use tiiuae/Falcon-H1-7B-Instruct to fine-tune on Tool calling, in an attempt to increase its performance on BFCL.

The Data used is a custom Dataset I made named younissk/tool-calling-mix. It is a mixture of different Tool Calling datasets, as well as synthetic data and non tool calling data, in an attempt to target Catastrophic Forgetting.

In total it is made up of a split of 60600 Train, 7580 Test and 7580 Eval rows, with a consistent percentage split of each category.

The Split is as follows:

4.1 Data sources and split

We train on a mixture of function-calling (single, multi, and parallel) and no-call language tasks to balance tool-use execution with general instruction-following and language understanding. Table 3 summarizes the sources, counts, and rationale.

Table 3: Training data split (total $N = 78{,}000$ examples).

Source	Rationale / coverage	Count	Share
XLAM/APIGen data [4, 9]	Verified function-calling (single & multi)	20,000	25.6%
Gorilla OpenFunctions [7]	Tool-call schemas and arguments	15,000	19.2%
ToolBench [8]	Multi-tool (rich multi-call trajectories)	20,000	25.6%
Dolly 15k (no-call) [1]	Instruction following without tools	8,000	10.3%
WikiText (no-call) [5]	General language understanding/modeling	8,000	10.3%
Synthetic Parallel (mine)	Parallel function-calling supervision	7,000	9.0%

4.2 Synthetic Data

We create a 7k-example synthetic set of *parallel tool-call* trajectories to strengthen multi-tool execution. The generator composes scenarios from multiple function categories and enforces the use of *at least two different tools* in each sample. The design is motivated by verifiable function-calling generation pipelines (e.g., APIGen) and multi-step tool-use prompting patterns (e.g., ReAct), while using our own templates and validation rules [4, 12, 7**?** , 8].

Pipeline. Given category-specific function schemas and prompt templates, we:

1. **Mix tools across categories.** Sample 2–3 tools from distinct categories to force parallelism (Algorithm: `EnhancedScenarioTemplates.get_function_templates()` + random mix).
2. **Prompt an LLM (if available).** Query a local `falcon-h1-34b` via a `llama.cpp`-compatible endpoint to produce a natural user request requiring *all* selected tools in parallel and to draft concrete calls (temperature 0.7, top-p 0.9). If the response is missing or malformed, fall back to deterministic templates.
3. **Canonicalize output.** Emit a record with:
   - `id`, `question` (chat-style messages), `function` (the available tools/schemas),
   - `ground_truth`: list of canonical call strings (e.g., `calculate_mean(numbers=[...])`),
   - `execution_result_type`: [‘‘exact_match’’, ...].
4. **Validate.** Enforce required fields, structure, and *at least two distinct tool names* in `ground_truth`; drop failures.
5. **Persist progressively.** Stream partial saves every 60s and a final JSON on completion.

Example (raw JSON). The following block shows the exact JSON structures for tools, messages, and a target with two parallel calls (as generated/collected before post-validation):

Listing 1: Example: tools, messages, and target JSON

```
{
  "tools_json": [
    {
      "name": "calculate_variance",
      "description": "Calculates variance of a dataset.",
      "parameters": {
        "type": "dict",
        "properties": {
          "data": {
            "type": "array",
            "items": {"type": "float"},
            "description": "The dataset."
          }
        },
        "required": ["data"]
      }
    },
```

```
    {
      "name": "calculate_standard_deviation",
      "description": "Calculates the standard deviation of a list of numbers.",
      "parameters": {
        "type": "dict",
        "properties": {
          "numbers": {
            "type": "array",
            "items": {"type": "float"},
            "description": "The list of numbers."
          }
        },
        "required": ["numbers"]
      }
    }
  ],

  "messages_json": [
    {
      "role": "user",
      "content": "For my project, I need to calculate variance and calculate
          standard deviation. Can you help with these calculations?"
    }
  ],

  "target_json": {
    "tool_calls": [
      {
        "name": "calculate_variance",
        "arguments": { "data": "[24.4" }
      },
      {
        "name": "calculate_standard_deviation",
        "arguments": { "numbers": "[1.6" }
      }
    ]
  }
}
```

Note. Our validator ensures well-formed arrays/arguments and multiple distinct tools per item; malformed items (e.g., truncated arrays) are rejected or auto-repaired by template fallback.

5 Training Setup

Objective. We fine-tune a pretrained Falcon-family model for reliable, structured tool-calling while minimizing memory and compute cost. The approach emphasizes parameter-efficient adaptation, conservative memory use, and stable optimization with early stopping.

Base model and numerical setup. The tokenizer and base checkpoint are loaded once; training runs either in a memory–efficient 4-bit configuration or in standard precision, depending on environment settings. On Ampere-class GPUs, TF32 is enabled for large matrix multiplies to improve throughput. Mixed precision is selected to match the model's dtype (bfloat16 or float16) when available. A CUDA allocator configuration favors expandable segments to reduce fragmentation.

Parameter-efficient adaptation. We perform Low-Rank Adaptation (LoRA) on a set of target modules that are inferred automatically if not specified. LoRA rank, scaling factor, and dropout are set from the experiment configuration.

Data pipeline and batching. Training/validation splits are prepared by a dataset loader that formats conversational samples for next-token prediction. Right-padding is used for inputs and labels; labels on padding positions are masked with an ignore index to exclude them from the loss. Attention

masks are derived from non-pad tokens. To stabilize step times and reduce padding waste, batches are formed with sequence-length grouping. Gradient checkpointing is enabled to lower activation memory.

Optimization and precision details. AdamW is our default optimizer. The learning-rate schedule is cosine with warmup; weight decay and gradient clipping are applied. Effective batch size is controlled via per-device batch size and gradient accumulation.

Training schedule and evaluation. Training runs for a fixed number of epochs with step-based validation. We checkpoint at fixed intervals with a cap on the number of retained checkpoints. The best model is selected by validation loss and restored at the end. Early stopping halts training when the validation metric plateaus beyond a configured patience. Logging occurs at regular step intervals, and an optional throughput hook prints tokens-per-second for quick diagnostics.

Optional weight interpolation (WiSE-FT style). When enabled, the adapter-merged model is linearly interpolated with the original base weights using a scalar coefficient and saved as an additional blended checkpoint. This provides a controllable trade-off between adapted behavior and base-model priors.

Systems and reproducibility notes. Environment and library versions (Python, PyTorch, CUDA availability, 4-bit mode) are printed at startup. If fast CUDA kernels for state-space modules are missing, a warning is emitted that training may be substantially slower on such hardware/software setups.

6 Results

The results underperformed the base models in all aspects except Parallel Tool calling. As it was trained on the 7B variant, direct comparison with the H1-7B results are the closest.

Table 4: Live evaluation results across all models.

Model	Live Acc	Live Simple AST	Live Multiple AST	Live Parallel AST
Falcon-H1-0.5B-Instruct (FC)	36.79%	34.50%	38.75%	0.00%
Falcon-H1-7B-Instruct (FC)	67.21%	70.54%	68.95%	0.00%
Falcon-H1-1.5B-Instruct (FC)	11.62%	0.00%	14.91%	0.00%
Falcon-H1-34B-Instruct (FC)	14.29%	74.81%	N/A	N/A
GPT-4o-2024-11-20 (FC)	13.47%	70.54%	N/A	N/A
Falcon Twig	24.50%	46.51%	19.75%	18.75%

An example of a failed parallel call is the following (Comparable to the other examples):

Failure Case: `live_parallel_0-0-0` — No acceptable match (schema mismatch)

Category: `live_parallel` **Valid:** No
Error Type: `parallel_function_checker_no_order:cannot_find_match`
Checker Message: "Could not find a matching function among index [0, 1] of model output." Missing required parameter: "location" (both calls).

Natural-language prompt (user):
"请问北京的当前天气状况如何？还有，上海的天气情况是怎样的？"

Tool schema (summary):

- `name: get_current_weather`
- `parameters: {location` (string, *required*), `unit` ("celsius" | "fahrenheit", default "fahrenheit")}
- *Format guidance:* If no state exists, use "City, Country" (e.g., "Beijing, China").

Expectations: Two *parallel* calls to `get_current_weather`:

- one for `Beijing, China`
- one for `Shanghai, China`

with `unit` either default/empty or "fahrenheit".

Model output (Actual; 2 calls, wrong schema)

```
[
  {
    "get_current_weather": {
      "city": "Beijing",
      "unit": "Celsius"
    }
  },
  {
    "get_current_weather": {
      "city": "Shanghai",
      "unit": "Celsius"
    }
  }
]
```

Accepted answers (Expected; 2 calls)

```
[
  { "get_current_weather": {
      "location": "Beijing, China",
      "unit": "fahrenheit" | ""
  }},
  { "get_current_weather": {
      "location": "Shanghai, China",
      "unit": "fahrenheit" | ""
  }}
]
```

The code and instructions are under `https://github.com/younissk/falcon-twig` and the dataset is under `https://github.com/younissk/tool-calling-mix`.

Additionally, the model is hosted on huggingface under `https://huggingface.co/younissk/Falcon-Twig-7B` and the dataset is under `https://huggingface.co/datasets/younissk/tool-calling-mix`

7 Discussion

7.1 Synthetic Data

Analyzing the synthetically generated parallel tool calling data, I see that some of it is wrong, meaning garbage in, garbage out. The synthetic data is also only in English, while adding multiple languages would be better.

The results produced by the LLM itself are also not the best. To truly fine-tune it on parallel tool calling, data quality is crucial and a thorough analysis of the data is needed.

7.2 Training methodolgy

The training process was optimized for efficiency, not results due to budget constraints.

Besides QLoRA, I propose using a form of Reinforcement learning, specifically for tool calling. However, this would require sufficient funds.

A meaningful test would be to try to fine tune a different LLM on the exact same data with the exact same methodology to rule out and problems with the training data or method.

References

[1] Databricks. Databricks Dolly 15k: An open instruction-following dataset. `https://huggingface.co/datasets/databricks/databricks-dolly-15k`, 2023. Accessed 2025-09-28.

[2] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. Qlora: Efficient finetuning of quantized llms. In *NeurIPS*, 2023.

[3] E. J. Hu and et al. Lora: Low-rank adaptation of large language models. *arXiv:2106.09685*, 2021.

[4] Z. Liu, T. Hoang, J. Zhang, M. Zhu, T. Lan, S. Kokane, J. Tan, W. Yao, Z. Liu, Y. Feng, R. Murthy, L. Yang, S. Savarese, J. C. Niebles, H. Wang, S. Heinecke, and C. Xiong. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024), Datasets and Benchmarks Track*, 2024. URL `https://proceedings.neurips.cc/paper_files/paper/2024/file/61cce86d180b1184949e58939c4f983d-Paper-Datasets_and_Benchmarks_Track.pdf`.

[5] S. Merity, C. Xiong, J. Bradbury, and R. Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations (ICLR)*, 2017. URL `https://openreview.net/forum?id=Byj72udxe`.

[6] A. omitted. Hammerbench: Fine-grained function-calling evaluation in multi-turn tasks. *arXiv:2412.16516*, 2025.

[7] S. G. Patil and et al. Gorilla: Large language model connected with massive tools. In *NeurIPS*, 2024.

[8] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, L. Hong, R. Tian, R. Xie, J. Zhou, M. Gerstein, D. Li, Z. Liu, and M. Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023. URL `https://arxiv.org/abs/2307.16789`.

[9] Salesforce AI Research. xLAM Function-Calling 60k dataset. `https://huggingface.co/datasets/Salesforce/xlam-function-calling-60k`, 2024. Accessed 2025-09-28.

[10] T. Schick and et al. Toolformer: Language models can teach themselves to use tools. *arXiv:2302.04761*, 2023.

[11] Q. Xu and et al. On the tool manipulation capability of open-source large language models. *arXiv:2305.16504*, 2023.

[12] S. Yao, J. Zhao, D. Yu, and et al. React: Synergizing reasoning and acting in language models. In *ICLR*, 2023.

[13] J. Zuo and et al. Falcon-h1: A family of hybrid-head language models redefining efficiency and performance. *arXiv:2507.22448*, 2025.