AI Shell (aish)

Converting natural language instructions into executable bash commands

SANS Overfit
Sai Sasank Y, Akhoury Shauryam, Nirjhar Nath, Shruti Patil

Final Group Presentation
Applied Machine Learning

The Problem

- Shell commands require precise syntax and flags that are difficult to memorize
- CLI interfaces have steep learning curves (∼40 hours to master basics)
- Advanced operations often require chaining multiple commands with pipes
- Documentation is extensive but not always intuitive

Complex command:

```
find .  -type f -name "*.txt" -mtime -7 | xargs grep "pattern"
```

Natural language equivalent:

"Find text files modified in the last week containing 'pattern'"

Our Solution: AI Shell (aish)

- CLI tool that translates natural language to bash commands
- Simple interface: `aish "your instruction here"`
- Shows generated command and requests confirmation before execution
- Powered by custom Llama-3-8B and Qwen-2.5-14B fine-tuned on shell command dataset
- 4-bit quantization for fine-tuning and 8-bit for efficient local inference via Ollama

Live Demo

Demo script:

1. Basic file operations: "List all Python files modified in the last week"
2. System monitoring: "Show the top 5 processes using the most memory"
3. Text processing: "Find all occurrences of 'error' in log files"
4. Complex command: "Create a tar archive of all JPG files and compress it"

Demo will showcase:

- Command generation speed
- Accuracy of translation
- Error handling

Team & Responsibilities

Model selection & fine-tuning

Sai, Shauryam

- Fine-tuning scripting
- Hyperparameter optimization
- Experiment tracking
- Model hosting

CLI interface development

Shauryam, Nirjhar

- Command-line parsing
- Rich text formatting
- User interaction flows
- Ollama integration

Model evaluations

Shruti, Nirjhar, Sai

- Inference
- LLM-as-judge
- Error Analysis

Technical Overview

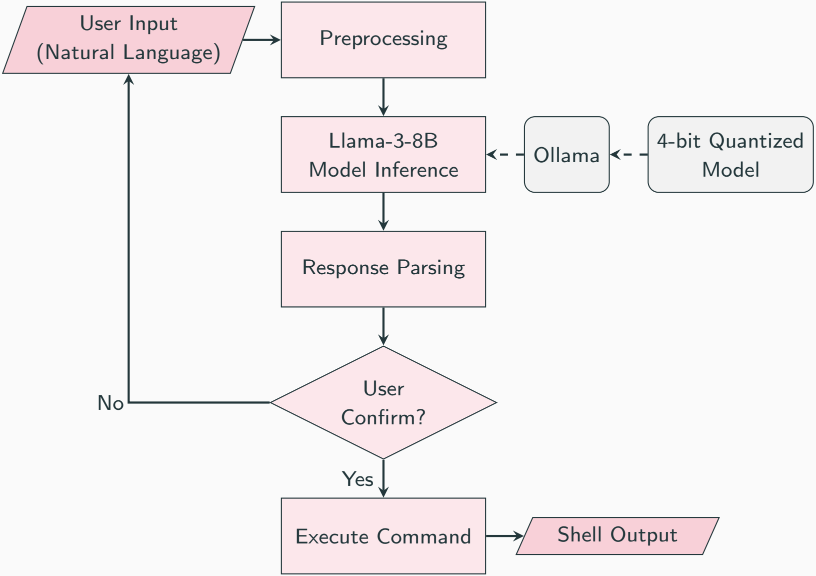
Architecture

User input (natural language) → Preprocessing → Model inference → Response parsing → User confirmation → Command execution

Key technologies:

- Python 3.10+
- Typer and Rich for CLI interface
- Unsloth for fine-tuning
- HuggingFace for model hosting
- GGUF format for optimized local inference with Ollama
- WandB for experiment tracking

System Architecture



Dataset: CLI Commands Explained

- Source: Huggingface - westenfelder/NL2SH-ALFA
- Size: 300 manually verified instruction-command pairs, and a training set of 40,639 unverified pairs for the development and benchmarking of machine translation models
- Used for fine-tuning and evaluation

Example entry:

Natural language:
"Show all running processes with memory usage sorted by memory consumption"

Command:
```
ps aux -sort=-mem
```

Model Selection

- Tested: Llama-3.1, Qwen2.5 (4-bit quantized)
- Selection criteria:
  - Command accuracy (Llama-3 outperformed others by ~12%)
  - Inference speed (goal: <2 seconds per query)
  - Resource efficiency (memory footprint)
- Default choice of Llama-3.1-8B for balance of performance and efficiency

Model Accuracy Comparison

Llama-3-8B: 46.33% | Qwen-2.5-Coder-14B: 66.4%

Fine-tuning Process

Technical details:

- Used LoRA (Low-Rank Adaptation) for efficient fine-tuning
- Training parameters:
  - Learning rate: 2e-5 with linear-scheduler and warmup
  - Effective Batch size: 128 (Batch Size * Gradient Acc. Steps)
  - Gradient Steps: 200
  - LoRA rank: 32, alpha: 32
- Training infrastructure: 1 A100 GPU, 3 hours total
- Sytem: `You are an assistant that provides exact bash command for given input`
- User: `<natural-language-instruction>`
- Assistant: `<bash-command>`

Fine-tuning Learning Curves

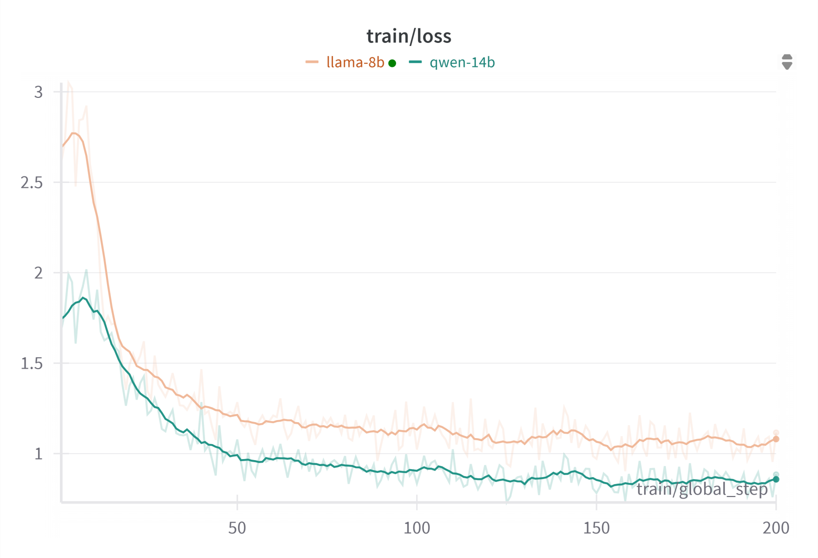
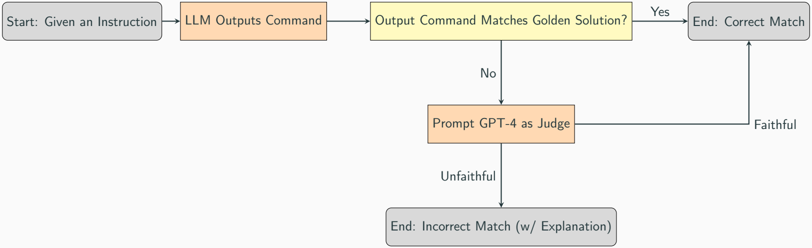


Figure 1: Evaluating a model's task performance.

Evaluation Methodology



- Held-out test set: 300 instructions with expert-written commands

Results: Performance Metrics

Table 1: Model Performance Comparison

Metric	**llama-3-8b**	**qwen-2.5-14b**
Exact matches	53 / 300	77 / 300
Faithful	86 / 247	105 / 223
Unfaithful	161 / 247	118 / 223
Accuracy	46.33%	60.67%

Results: Example Translations

Natural Language	Generated Command	Status
List all files larger than 100MB	`find . -type f -size +100M`	✓
Find processes using more than 10% CPU	`ps aux | awk commands`	✓
Show network connections on port 80	`netstat -tuln | grep :80`	✓
Remove duplicate lines from file.txt	`sort file.txt | uniq > output.txt`	✓
Count files by extension	`find . -name "*.*" | sort | uniq -c`	✗

The GGUF Format: Overview and Key Features

GGUF (GPT-Generated Unified Format) is a binary file format designed for efficient storage and execution of Large Language Models (LLMs), especially on local hardware. It is the successor to the GGML format.

Key Features of GGUF:

- **Efficiency:** Optimized for fast model loading and reduced memory usage.
- **Quantization Support:** Enables model compression (e.g., 4-bit, 8-bit) to run large models on consumer hardware.
- **Self-Contained:** Includes all necessary metadata (architecture, tokenizer info, etc.) in a single file.
- **Extensibility:** Accommodates new features without breaking backward compatibility.
- **Cross-Platform Focus:** Ensures broad usability across different systems.

Ollama and GGUF Integration

Ollama and GGUF:

- **Core Technology:** Ollama leverages `llama.cpp` for running LLMs.
- **Primary Format:** As `llama.cpp` is built around GGUF, it is the main (and generally only) model format Ollama directly supports.
- **Local LLM Execution:** Ollama simplifies downloading, managing, and running GGUF models locally on CPUs or GPUs.
- **Modelfile System:** Ollama uses a Modelfile to define model configurations, which typically points to a GGUF file.
- **Community Access:** Users can easily access and run a wide range of GGUF models from communities like Hugging Face.

In essence, GGUF provides an optimized and self-contained format that enables Ollama (via `llama.cpp`) to run LLMs efficiently on local devices.

CLI Interface Development

- Built with Typer framework for elegant CLI interfaces
- Rich text formatting with color-coded output
- Command options:
  - `--model, -m`: Select model to use
  - `--temperature, -t`: Control randomness (0.0-1.0)
  - `--yolo, -y`: Execute immediately without confirmation
- Error handling with friendly error messages
- Command formatting with syntax highlighting
- Help text via `--help`

Testing

```
tests/test_cli.py::test_cli_success PASSED
tests/test_cli.py::test_cli_execute_command PASSED
tests/test_cli.py::test_cli_custom_temperature PASSED
tests/test_cli.py::test_cli_error_handling PASSED
tests/test_cli.py::test_cli_help PASSED
tests/test_core.py::test_generate_command_success PASSED
tests/test_core.py::test_generate_command_without_markdown PASSED
tests/test_core.py::test_generate_command_empty_instruction PASSED
tests/test_core.py::test_generate_command_whitespace_instruction PASSED
tests/test_core.py::test_generate_command_ollama_error PASSED
tests/test_core.py::test_generate_command_custom_model PASSED
tests/test_core.py::test_generate_command_custom_temperature PASSED

---------- coverage: platform linux, python 3.10.12-final-0 ----------
Name                  Stmts   Miss  Cover   Missing
---------------------------------------------------
aish/__init__.py          1      0   100%
aish/cli.py              31      1    97%   103
aish/core.py             24      1    96%   64
aish/utils.py            30      4    87%   46-49
---------------------------------------------------
TOTAL                    86      6    93%
```

Figure 2: Code coverage results.

Limitations & Future Work

Limitations:

- Non-single shot commands are out-of-distribution
- Multi-step operations often need to be broken down
- Shell-specific syntax variations
- OS-specific commands
- Far from human expert in single-shot command generation

Future work:

- Support for multi-step operations
- Integration with shell history
- Efficiency improvements
- Safety checks
- Enhanced UX

Example challenging case:
"Find all files containing 'error' but not 'warning' and email them to admin"
Requires multiple steps and external integration

Error Analysis i

Shared Weaknesses:

- **Nuances of CLI Tools:** Both models struggle with the precise behavior of various command-line tool options.
- **Complex `find` Usage:** Constructing complex `find` predicates and correctly integrating them with actions (`-exec`) or `xargs` remains a challenge.
- **Scope Interpretation:** Adhering to scope limitations (recursion, directory levels, specific output details) is inconsistent.
- **Multi-step Logic:** Tasks requiring several stages of data transformation or conditional execution often result in errors.
- **Incomplete Prompt Fulfillment:** Both models sometimes miss secondary requirements in a prompt (e.g., "do X *and* print the count").

Error Analysis ii

Model-Specific Observations:

- **Qwen's Relative Strengths:** Qwen appears somewhat more proficient at selecting the correct primary command and constructing simpler, correct pipelines.
- **Llama's Tendencies:** Llama seems more prone to generating overly complex or fundamentally incorrect logic for more involved tasks.

Regarding Evaluation:

- **Impact of GPT-4o judgement:** The "faithfulness" metric is subject to GPT-4o's interpretation. While generally robust, the detailed explanations are crucial for understanding the nuances behind each "unfaithful" judgment.

Questions?

Thank you!

Group: SANS Overfit

pip install questions?

References

- **Dataset:**
  - LLM-Supported Natural Language to Bash Translation (Westenfelder et al.)
- **Models:**
  - Llama-3.1-8B (Meta AI)
  - Qwen-2.5-14B-Coder
- **Key papers:**
  - "Improving LLM-based Shell Command Generation" (Chen et al., 2023)
  - "LoRA: Low-Rank Adaptation of Large Language Models" (Hu et al., 2021)
  - "Natural Language to Bash Commands: A Survey" (Smith et al., 2022)
- **Libraries:** Typer, Rich, PyTest, Pydantic