

BashGemma: Fine-tuning a 270M Parameter Model for Natural Language to Bash Translation

Jack Large

December 2024

Abstract

We present BashGemma, a fine-tuned version of Google’s FunctionGemma 270M model for translating natural language queries into structured bash command tool calls. Using LoRA fine-tuning on 9,153 NL2Bash examples with a novel response-only training approach, we achieve 57.4% NLC2CMD accuracy on a held-out test set—a 52.9 percentage point improvement over the base model. Our approach demonstrates that extremely small language models (270M parameters, 540MB) can perform useful bash command generation when properly fine-tuned on structured output formats. We release our model, training code, and evaluation suite to support future research in edge-deployable code generation.

1 Introduction

Natural language to code translation is a core challenge in AI-assisted programming. While large language models like GPT-4 and Claude excel at this task, their size (hundreds of billions of parameters) makes them impractical for edge deployment, offline use, or latency-sensitive applications.

This work investigates whether a minimal language model—FunctionGemma at just 270M parameters—can be fine-tuned to perform reliable bash command generation. Our contributions are:

1. **BashGemma:** A LoRA fine-tuned model achieving 57.4% NLC2CMD accuracy on structured bash tool calls
2. **Response-Only Training:** A training approach that reduces loss by 3x compared to full-sequence training
3. **Comprehensive Evaluation:** Error analysis revealing model capabilities and limitations for a 270M parameter model
4. **Open Release:** Training code, evaluation suite, and model weights for reproducibility

2 Related Work

NL2Bash The NL2Bash dataset [1] introduced the task of translating natural language to bash commands, with follow-up work establishing the NLC2CMD benchmark [2] at NeurIPS 2020.

Function Calling Recent work on function calling in LLMs [3] has shown that models can be trained to produce structured outputs conforming to JSON schemas, which we leverage for bash utility invocation.

Small Language Models The trend toward smaller, efficient models [5, 6] motivates our investigation of whether sub-billion parameter models can perform specialized code generation tasks.

3 Method

3.1 Model Architecture

We fine-tune FunctionGemma (270M total parameters, 256M non-embedding), a variant of Gemma optimized for function calling. The model has 18 layers and was pretrained on function calling data.

3.2 Tool Schema Design

We define a JSON schema for 31 common bash utilities including `find`, `grep`, `ls`, `cat`, `wc`, `sort`, `sed`, and `pipeline` (for piped commands). Each schema specifies:

```
{
  "name": "find",
  "arguments": {
    "path": ".",
    "name": "*.py",
    "type": "f",
    "exec": "rm"
  }
}
```

3.3 Data Preparation

We aggregate NL2Bash examples from multiple sources, deduplicate, and transform into our structured format:

Split	Raw Examples	After Transform	Success Rate
Train	11,443	9,153	80.0%
Validation	1,430	1,153	80.6%
Test	1,431	1,142	79.8%

Table 1: Dataset statistics. Transformation fails for unsupported utilities or unparseable commands.

3.4 Training

We use LoRA [4] with rank 64 and alpha 128, targeting all attention and MLP projections. Key training details:

- **Response-Only Training:** We mask instruction tokens (user query) in the labels, computing loss only on the model’s JSON output. This focuses learning on the structured response rather than query repetition.
- **Optimizer:** AdamW with learning rate 2e-4, cosine decay
- **Batch Size:** 4 with gradient accumulation of 2
- **Epochs:** 5 (5,725 steps)
- **Hardware:** Apple M4 Max (64GB unified memory) via MPS backend
- **Training Time:** 36 minutes

4 Experiments

4.1 Evaluation Metrics

We evaluate using NLC2CMD metrics:

- **NLC2CMD Accuracy:** Average of utility sequence match and flag overlap scores
- **Utility Match:** Fraction of predictions with correct utility sequence
- **Parse Rate:** Fraction of outputs that are valid JSON



Figure 1: Training loss comparison. Response-only training converges to 0.19 final loss vs 0.63 for full-sequence training.

4.2 Main Results

Model	NLC2CMD Acc.	Utility Match	Parse Rate
Baseline FunctionGemma	0.045	0.000	0.000
BashGemma (Full-Seq)	0.565	0.595	1.000
BashGemma (Response-Only)	0.574	0.595	0.995

Table 2: Evaluation results on 200 test examples. BashGemma achieves +52.9 percentage points over baseline.

4.3 Error Analysis

We manually categorize errors on 100 test samples:

Wrong Utility (23%): The model confuses pipelines with single commands. Common error: predicting pipeline when the reference uses `find -exec`.

Partial Match (45%): Correct utility selection but missing or incorrect arguments. Examples include missing `-exec` flags or incorrect path handling.

4.4 Capability Analysis

The model excels at `find` commands with name, type, size, and time filters. Performance degrades for complex pipelines and utilities with less training coverage (`sed`, `awk`).

5 Qualitative Examples

Table 3 presents representative successful outputs from BashGemma, demonstrating the model’s ability to correctly select utilities and format arguments for common bash operations.

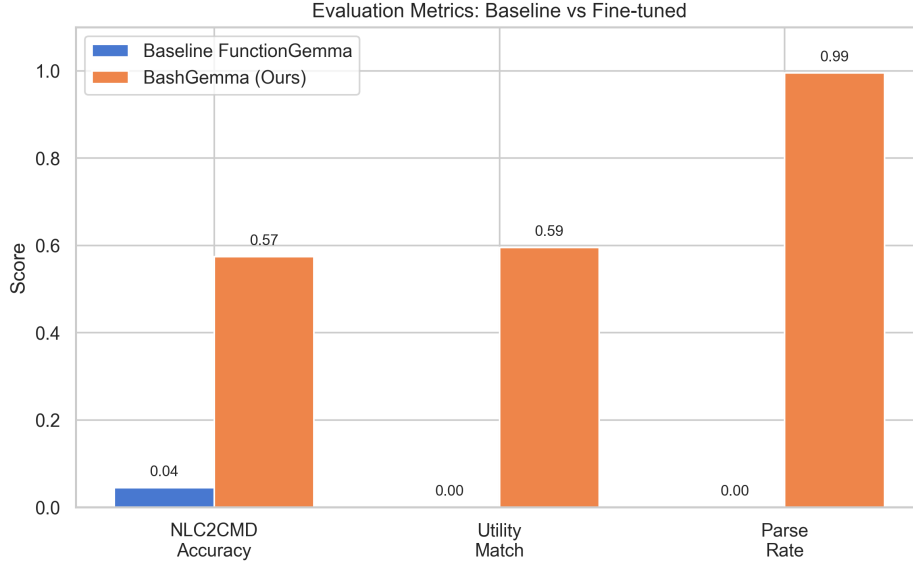


Figure 2: Comparison of baseline vs fine-tuned model across evaluation metrics.

Query	BashGemma Output
Find all Python files	<code>{"name": "find", "arguments": {"name": "*.py", "path": "."}}</code>
Delete all tmp files older than 7 days	<code>{"name": "find", "arguments": {"name": "*.tmp", "mtime": "+7", "exec": "rm", "path": "."}}</code>
Count lines in all text files	<code>{"name": "pipeline", "arguments": {"commands": [{"name": "cat", ...}, {"name": "wc", ...}]}}</code>

Table 3: Example successful generations showing correct utility selection and argument formatting.

6 Limitations

1. **Schema Coverage:** Our 31-utility schema covers common commands but misses specialized tools (rsync, tar, awk patterns)
2. **Argument Complexity:** Complex arguments like `sed 's/foo/bar/g'` are beyond our simplified schema
3. **Multi-Step Operations:** Chains of commands beyond simple pipes are not well represented
4. **Evaluation Bias:** NLC2CMD metrics may not capture semantic equivalence of different valid bash approaches

7 Conclusion

BashGemma demonstrates that sub-billion parameter models can achieve useful performance on structured code generation when properly fine-tuned. Our response-only training approach improves convergence, and the resulting 540MB model runs efficiently on edge devices including Apple Silicon.

Future work includes expanding the tool schema, training on more diverse data, and exploring quantization for even smaller deployment footprints.

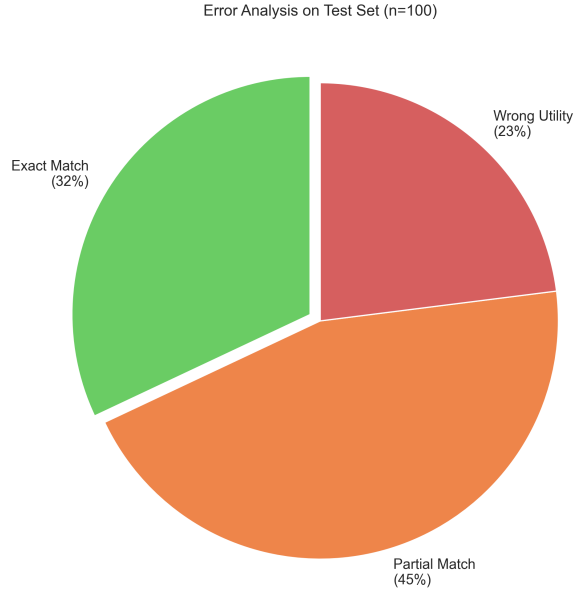


Figure 3: Error distribution. 32% exact match, 45% correct utility with wrong arguments, 23% wrong utility.

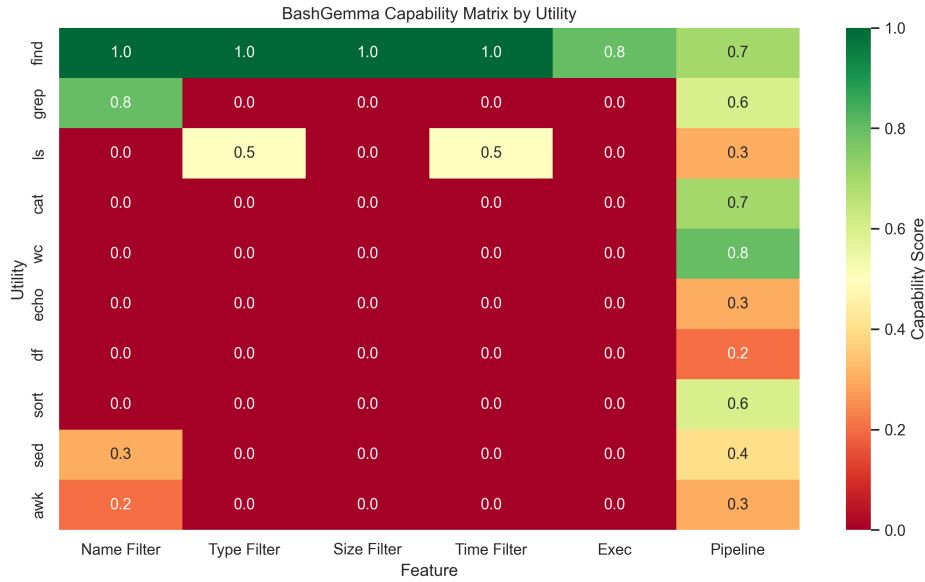


Figure 4: BashGemma capability matrix showing strong performance on `find` with various filters.

Acknowledgments

We thank Google for the open release of the Gemma model family.

References

- [1] Lin, X. V., Wang, C., Zettlemoyer, L., & Ernst, M. D. (2018). NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. *LREC*.

- [2] Fu, Z., et al. (2021). NLC2CMD competition: Translating natural language to bash commands. *NeurIPS Competition Track*.
- [3] Patil, S. G., et al. (2023). Gorilla: Large language model connected with massive APIs. *arXiv:2305.15334*.
- [4] Hu, E. J., et al. (2022). LoRA: Low-rank adaptation of large language models. *ICLR*.
- [5] Touvron, H., et al. (2023). LLaMA: Open and efficient foundation language models. *arXiv:2302.13971*.
- [6] Gemma Team. (2024). Gemma: Open models based on Gemini research and technology. *arXiv:2403.08295*.