Fine-Tuning a Large Language Model for AWS IAM Policy Generation

Converting Natural Language Descriptions to Valid IAM JSON Policies

Author: Vatsal Naik

Abstract

This report presents the development of a fine-tuned Mistral-7B model that converts natural language security requirements into valid AWS IAM JSON policies. Using QLoRA (4-bit quantized Low-Rank Adaptation) for parameter-efficient fine-tuning on a curated dataset of 1,488 examples, the model achieves a 60.3% JSON validity rate compared to 0.7% for the base model in zero-shot mode. We evaluate four hyperparameter configurations, perform a three-way baseline comparison (zero-shot vs. few-shot vs. fine-tuned), and conduct detailed error analysis by policy complexity. The fine-tuned model demonstrates particular strength on simple policies (76.2% accuracy) while identifying clear improvement paths for complex multi-service policies.

Table of Contents

1. Introduction
2. Dataset Preparation
3. Model Selection & Architecture
4. Fine-Tuning Setup
5. Hyperparameter Optimization
6. Model Evaluation
7. Error Analysis
8. Inference Pipeline
9. Ethical Considerations & Limitations
10. Conclusion & Future

Work References

1. Introduction

AWS Identity and Access Management (IAM) policies are JSON documents that define permissions for AWS resources. Writing correct IAM policies is a common pain point for cloud engineers: the policies must follow a strict schema, use exact AWS action names, and correctly specify Amazon Resource Names (ARNs). Errors in IAM policies can lead to either over-permissive access (security risk) or blocked operations (productivity loss).

This project addresses this challenge by fine-tuning a large language model to automatically convert natural language descriptions of security requirements into valid, correctly-structured IAM policies. For example, given the input "Allow read-only access to S3 bucket customer-data and write logs to CloudWatch," the model generates a complete, valid IAM policy JSON with the correct actions, resources, and ARN formats.

1.1 Motivation

- IAM policy authoring is error-prone and requires memorizing hundreds of AWS action names
- Existing tools (AWS Policy Generator) require manual clicking through UI forms
- LLMs have shown promise in code generation but struggle with strict JSON schema requirements
- A specialized fine-tuned model could serve as a developer productivity tool

1.2 Approach

We use Mistral-7B-v0.3 as the base model with QLoRA (4-bit quantized Low-Rank Adaptation) for memory-efficient fine-tuning. The model is trained on 1,189 instruction-response pairs using the Alpaca format, then evaluated against zero-shot and few-shot baselines on a held-out test set of 151 examples.

2. Dataset Preparation

2.1 Data Sources

The training dataset was curated from two complementary sources to ensure both breadth and quality:

Source 1: AWS Managed Policies (1,417 examples)

We programmatically pulled all AWS managed policies using the boto3 API (aws iam list-policies, aws iam get-policy-version). Policy names were converted to natural language instructions using CamelCase splitting (e.g., AmazonS3ReadOnlyAccess becomes "Provide an IAM policy for Amazon S3 Read Only Access"). Policies with more than 50 statements were excluded as they exceed reasonable generation length.

Source 2: Hand-Crafted Synthetic Examples (71 examples)

To address gaps in the managed policy dataset, we created 71 synthetic examples covering: diverse writing styles (casual to formal), conditional policies (MFA, IP restrictions, VPC endpoints, time-based), explicit deny policies, multi-service combinations, and tag-based access control. These examples ensure the model learns to handle varied input styles and advanced IAM features.

2.2 Preprocessing

- JSON validation: Every policy parsed and verified as valid JSON
- IAM schema check: Version, Statement array, Effect/Action/Resource per statement
- Deduplication: MD5 hash of normalized JSON to remove 20 exact duplicates

• Final validated count: 1,488 examples from 1,508 raw examples (98.7% pass rate)

2.3 Data Split

Data was split 80/10/10 with stratification by complexity (simple, medium, complex) to ensure balanced representation across all splits:

Split	Count	Percentage
Training	1,189	80%
Validation	148	10%
Test	151	10%
Total	1,488	100%

Table 1: Dataset split distribution

2.4 Data Format

Each example is formatted in the Alpaca instruction format, where the instruction contains the natural language description and the response contains the IAM policy JSON. Complexity is classified based on statement count and number of distinct AWS services: simple (1 statement, 1 service), medium (2-3 statements or 2-3 services), complex (4+ statements or 4+ services, or conditional logic).

3. Model Selection & Architecture

3.1 Base Model: Mistral-7B-v0.3

We selected Mistral-7B-v0.3 as the base model for the following reasons:

- **JSON generation capability**: Strong performance on structured output generation tasks
- **Size-performance tradeoff**: 7B parameters provides sufficient capacity while remaining trainable on a single GPU
- **Open access license**: Apache 2.0 license allows unrestricted use and modification
- **Community support**: Extensive documentation and integration with Hugging Face ecosystem

We considered but rejected several alternatives. BERT-family models are encoder-only and cannot perform autoregressive generation. GPT-2 (1.5B) has insufficient capacity for complex JSON structures. Larger models (LLaMA-70B, Mixtral-8x7B) exceed single-GPU memory even with quantization. CodeLlama-7B was a viable alternative but Mistral-7B showed stronger general instruction-following capability in our initial experiments.

3.2 QLoRA: 4-Bit Quantized Fine-Tuning

To fit the 7B parameter model on a single A100 GPU and enable efficient fine-tuning, we employ QLoRA (Quantized Low-Rank Adaptation). The base model weights are quantized to 4-bit NF4 (Normal Float 4) format, reducing memory from approximately 14GB to 4GB. Low-Rank Adaptation (LoRA) inserts small trainable adapter matrices into the model's attention and MLP layers, training only 0.6% of total parameters.

Parameter	Value
Quantization	4-bit NF4 with double quantization
Compute dtype	bfloat16
LoRA rank (r)	16
LoRA alpha	32
Target modules	q, k, v, o, gate, up, down projections
LoRA dropout	0.05
Trainable parameters	41.9M (0.58% of 7.29B total)
GPU memory usage	4.1 GB for model weights

Table 2: QLoRA and LoRA configuration (Config A)

4. Fine-Tuning Setup

4.1 Training Environment

Component	Specification
GPU	NVIDIA A100-SXM4-80GB
CUDA Version	12.1
Framework	PyTorch 2.5.1 + Hugging Face Transformers
Trainer	TRL SFTTrainer (Supervised Fine-Tuning)
Precision	bfloat16 mixed precision

Table 3: Training environment

4.2 Training Configuration

Training uses the SFTTrainer from the TRL library with a cosine learning rate schedule, warmup period, and early stopping with patience of 3 evaluation steps. Checkpoints are saved every 50 steps with a maximum of 3 retained. The best model is selected based on minimum validation loss.

Parameter	Value
Epochs	3
Batch size	4 per device
Gradient accumulation	4 steps (effective batch = 16)
Learning rate	2e-4
LR scheduler	Cosine
Warmup ratio	5%
Weight decay	0.01
Max sequence length	1,024 tokens
Early stopping patience	3 evaluations
Training time	~23 minutes

Table 4: Training hyperparameters (Config A)

4.3 Logging & Checkpointing

Training metrics are logged every 10 steps, with evaluation on the validation set every 50 steps. All training logs are persisted as JSON files for post-hoc analysis and visualization. The complete training history enables plotting loss curves and identifying optimal stopping points.

5. Hyperparameter Optimization

We trained four configurations varying learning rate, LoRA rank, LoRA alpha, and number of epochs to identify the optimal setup:

Config	LR	LoRA r	Alpha	Epochs	Train Loss	Best Eval Loss
A (best)	2e-4	16	32	3	0.1522	0.3019
B	1e-4	32	64	3	0.1584	0.3025
C	5e-4	16	32	5	0.0481	0.3207
D	2e-4	8	16	3	0.1887	0.3114

Table 5: Hyperparameter comparison across 4 configurations

5.1 Analysis

Config A (winner): The balanced configuration achieved the lowest validation loss (0.3019). The learning rate of 2e-4 with LoRA rank 16 provides sufficient model capacity without overfitting.

Config B (close second): Doubling the LoRA rank to 32 with a halved learning rate produced nearly identical results (eval loss 0.3025), suggesting the dataset size does not benefit from additional adapter capacity.

Config C (overfitting): The aggressive learning rate (5e-4) with 5 epochs caused severe overfitting. Training loss dropped to 0.048 but validation loss increased after step 200, reaching 0.3207. Early stopping triggered at epoch 4.

Config D (underfitting): The minimal LoRA rank (r=8) provided insufficient capacity, resulting in the highest validation loss (0.3114) and highest training loss (0.1887) among all configurations.

6. Model Evaluation

6.1 Evaluation Metrics

We evaluate five complementary metrics:

- **JSON Valid Rate**: Percentage of outputs that are directly parseable as valid JSON (no extraction needed)
- **JSON Extracted Rate**: Percentage where valid JSON can be found within the output (may include surrounding text)
- **Schema Valid Rate**: Percentage with correct IAM structure (Version, Statement, Effect, Action, Resource)
- **Service Accuracy**: Jaccard similarity of AWS service prefixes between generated and expected policies
- **Effect Accuracy**: Whether the Allow/Deny effects match the expected policy

6.2 Three-Way Baseline Comparison

All three approaches are evaluated on the identical 151-example test set:

Metric	Zero-Shot	Few-Shot	Fine-Tuned
JSON Valid Rate	0.7%	2.0%	60.3%
JSON Extracted Rate	79.5%	7.3%	60.3%
Schema Valid Rate	78.8%	7.3%	60.3%

Metric	Zero-Shot	Few-Shot	Fine-Tuned
Service Accuracy	28.7%	0.0%	52.0%
Effect Accuracy	98.3%	100.0%	95.6%

Table 6: Three-way evaluation comparison on 151 test examples

6.3 Key Findings

The fine-tuned model achieves a **60.3% JSON valid rate** compared to just 0.7% for zero-shot. While zero-shot shows a high "extracted" rate (79.5%), this is because it wraps JSON inside markdown code blocks (```json...```) which requires post-processing to extract. The fine-tuned model produces clean, directly-parseable JSON output suitable for production use.

Few-shot performance is surprisingly poor (7.3% schema valid) because the 3 in-context examples consume most of the context window, causing truncated outputs. This demonstrates that fine-tuning is significantly more effective than in-context learning for this structured generation task.

6.4 Performance by Complexity

Complexity	Zero-Shot	Few-Shot	Fine-Tuned
Simple	33/42 (79%)	3/42 (7%)	32/42 (76%)
Medium	34/39 (87%)	5/39 (13%)	26/39 (67%)
Complex	52/70 (74%)	3/70 (4%)	33/70 (47%)

Table 7: Schema valid rate by policy complexity

The fine-tuned model shows a clear inverse relationship between complexity and accuracy: simple policies achieve 76.2% correctness while complex policies drop to 47.1%. This is primarily due to output truncation on longer policies rather than structural errors.

7. Error Analysis

7.1 Error Categorization

Error Category	Count	Rate	Root Cause
Correct	44	29.1%	Valid JSON + correct schema + right services
Truncated (No JSON)	60	39.7%	Complex policies exceed max_new_tokens
Wrong Services	43	28.5%	Model approximates but misses exact action names
Wrong Effect	4	2.6%	Rare Allow/Deny confusion
Invalid Schema	0	0.0%	Model consistently produces correct structure

Table 8: Fine-tuned model error breakdown on 151 test examples

7.2 Pattern Analysis

Truncation errors (39.7%) are the dominant failure mode. These occur when the expected policy has many statements (4+) and the generation reaches the maximum token limit before completing the JSON structure. The generated content up to the truncation point is typically correct, suggesting the model has learned the right patterns but needs more generation budget for complex policies.

Wrong service errors (28.5%) occur when the model uses related but incorrect AWS actions. For example, generating waf:* and waf-regional:* for a WAF policy when the expected output also includes wafv2:* actions. This reflects the model learning general patterns but lacking precise knowledge of all AWS service-action mappings.

Zero invalid schema errors (0%) is a notable success: the model never produces structurally broken policies. Every generated JSON that parses correctly also has proper Version, Statement, Effect, Action, and Resource fields.

7.3 Suggested Improvements

- **Constrained decoding**: Use grammar-guided generation to enforce valid JSON at every token
- **Post-processing validation**: Add a validation layer that auto-corrects common schema issues and truncation
- **RAG with AWS action catalog**: Retrieve valid action names from an AWS service index to prevent hallucination
- **Larger dataset**: Expand to 5,000+ examples with emphasis on complex multi-service policies
- **Increased max_new_tokens**: Allow 1024+ tokens for complex policies (trading inference speed)
- **DPO/RLHF**: Use preference optimization to improve semantic correctness beyond supervised fine-tuning

8. Inference Pipeline

We provide two inference interfaces for the fine-tuned model:

8.1 Python Pipeline Class

A reusable IAMPolicyGenerator class (inference_pipeline.py) that loads the model and provides a simple generate() method. The class handles tokenization, generation with configurable parameters (temperature, max tokens), response parsing, and JSON validation. This is suitable for programmatic integration.

8.2 Gradio Web Interface

A Gradio-based web application (inference_app.py) that provides an interactive browser interface with text input for the policy description, adjustable generation parameters (max tokens, temperature), syntax-highlighted JSON output, and real-time validation status. Pre-built examples demonstrate common use cases.

9. Ethical Considerations & Limitations

9.1 Security Implications

Generated IAM policies should always be reviewed by a security engineer before deployment. The model may generate over-permissive policies (using wildcards like Resource: "*" when specific ARNs are more appropriate) or miss important deny statements. The tool is intended to assist policy authoring, not replace human review.

9.2 Dataset Bias

The training data is predominantly sourced from AWS managed policies, which follow specific naming conventions and patterns. The model may underperform on unconventional descriptions or domain-specific terminology not represented in the training data. Additionally, the managed policies may not reflect the latest AWS services or actions released after the data collection date.

9.3 Limitations

- Complex multi-service policies (4+ statements) have lower accuracy (47.1%)
- Long policies may be truncated during generation
- Model may hallucinate uncommon or non-existent AWS action names
- Training data does not cover all possible IAM policy patterns or edge cases
- Model requires GPU for inference (not suitable for serverless deployment without optimization)

10. Conclusion & Future Work

This project demonstrates that parameter-efficient fine-tuning of a 7B-parameter language model can significantly improve IAM policy generation compared to zero-shot and few-shot approaches. The fine-tuned model achieves 60.3% JSON validity rate (vs. 0.7% zero-shot), 52% service accuracy (vs. 28.7%), and 0% structural errors. Training required only 23 minutes on a single A100 GPU with QLoRA reducing the trainable parameters to 0.58% of the total model.

The clear error categorization reveals that the primary bottleneck is output truncation for complex policies, not model capability. This suggests that increasing generation length and dataset size for complex examples would yield the most impactful improvements. The addition of constrained decoding and RAG with an AWS action catalog could further push accuracy toward production-grade levels.

Future work includes expanding the dataset to 5,000+ examples with emphasis on complex policies, implementing constrained JSON decoding, building a retrieval-augmented pipeline with the AWS service authorization reference, and exploring model distillation to enable deployment on smaller GPU instances or CPU-only environments.

References

[1] Dettmers, T., et al. (2023). QLoRA: Efficient Finetuning of Quantized Language Models. NeurIPS 2023.

[2] Hu, E., et al. (2022). LoRA: Low-Rank Adaptation of Large Language Models. ICLR 2022.

[3] Jiang, A., et al. (2023). Mistral 7B. arXiv:2310.06825.

[4] AWS Documentation. IAM Policy Reference. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies.html

[5] Hugging Face. PEFT: Parameter-Efficient Fine-Tuning. https://github.com/huggingface/peft

[6] Hugging Face. TRL: Transformer Reinforcement Learning. https://github.com/huggingface/trl

[7] Wang, Y., et al. (2023). Self-Instruct: Aligning Language Models with Self-Generated Instructions. ACL 2023.

[8] Taori, R., et al. (2023). Stanford Alpaca: An Instruction-following LLaMA Model. GitHub.