Ollama Cache System Explanation

What Are the 20,000 Ollama Calls?

The Problem

- We have **5,000 jobs** (HDFS log entries)
- We have **4 prompt templates** (simple, standard, fewshot_1, fewshot_3)
- Total combinations: **5,000 × 4 = 20,000 job-prompt pairs**

Each pair represents: "What does the LLM predict for THIS specific job using THIS specific prompt?"

What Gets Generated

For EACH of the 20,000 pairs, we need to know:

```
{
    "job": "081109 203518 148 INFO dfs.DataNode: PacketResponder 2 terminating",
    "prompt": "fewshot_3",
    "llm_prediction": 0,  # LLM says "Normal"
    "ground_truth": 0,    # Actual label is "Normal"
    "success": True       # LLM was correct! ✓
}
```

Current State (5% Real Data)

Currently in `predictors/mlbp.py`:

```
GROUNDING_RATIO = 0.05        # Only 5% real
MAX_GEMINI_CALLS = 2000       # Max 2000 calls
```

This means:

- ☑ **1,000 pairs (5%)**: Real Ollama API calls → cached to disk
- ✗ **19,000 pairs (95%)**: Fake heuristic simulation
- ⚠ **Problem**: XGBoost learns from 95% synthetic data!

Target State (100% Real Data)

After running `generate_real_data.py`:

```
GROUNDING_RATIO = 1.0         # Use 100% real
MAX_GEMINI_CALLS = 20000      # All 20k calls
```

This means:

- ☑ **20,000 pairs (100%)**: Real Ollama API calls → cached to disk
- ✖ **0 pairs (0%)**: No synthetic data
- ☑ **Solution**: XGBoost learns from 100% real LLM behavior!

Where Are Results Saved?

Cache Directory Structure

```
results/ollama_cache/
├── 039fe878fbf8a00d201af5db26858476.json  ← Job #1 + simple prompt
├── 057e7292542ce93939d3e726390a42dc.json  ← Job #1 + standard prompt
├── 07d29317a0c52bbc55c335c2efd6ae5e.json  ← Job #1 + fewshot_1 prompt
├── 0baeea4716067b3b4dbf056270de11bf.json  ← Job #1 + fewshot_3 prompt
├── ...
└── ffee87ac39e9043ac1dca2fafa49526ae.json ← Job #5000 + fewshot_3
```

Total files after full generation: ~20,000 JSON files

Cache File Format

Each JSON file contains:

```json
{
  "label": 0,
  "response": "Normal",
  "model": "gemma3",
  "timestamp": "2026-01-19T12:34:56"
}
```

Cache Key Generation

In `llm/gemini_client.py`:

```python
def _hash(job_text, prompt_name):
    key = f"{prompt_name}::{job_text}"
    return hashlib.md5(key.encode()).hexdigest()
```

Example:

- Job text: `"081109 203518 148 INFO dfs.DataNode..."`
- Prompt: `"fewshot_3"`
- Hash: `039fe878fbf8a00d201af5db26858476`
- File: `results/ollama_cache/039fe878fbf8a00d201af5db26858476.json`

How Will Cache Be Used?

Phase 1: Generation (One-Time Cost)

Run once to populate cache:

```
python generate_real_data.py 5000
```

Time: ~1-2 hours (20,000 Ollama API calls)

Phase 2: Training (Fast, Repeatable)

After cache is ready:

```
# predictors/mlbp.py
GROUNDING_RATIO = 1.0  # ← Change this to 1.0

# Then run:
python main.py
```

What happens:

1. `create_training_data()` loops through 20,000 job-prompt pairs
2. For each pair, calls `query_gemini(job, prompt)`
3. `query_gemini()` **checks cache FIRST**:

   ```
   if os.path.exists(cache_path):
       with open(cache_path, "r") as f:
           return json.load(f)["label"]  # ← Returns immediately!
   ```

4. Since ALL 20,000 are cached → **NO NEW API CALLS**
5. Training data is 100% real LLM predictions
6. XGBoost trains on realistic success/failure patterns

Time: ~10 seconds (just file I/O, no API calls)

Benefits of Full Cache

Metric	Before (5% Real)	After (100% Real)
Training Data Quality	95% synthetic	100% real
Model Accuracy	~79%	Target 85%+
API Calls Per Run	1,000	0 (cached)

Metric	Before (5% Real)	After (100% Real)
Training Time	60 seconds	10 seconds
Experiment Iterations	Slow (API wait)	Fast (instant)

The Full Workflow

1. Initial Setup (Current State)

```
Jobs (5000) × Prompts (4) = 20,000 pairs needed
   ↓
Only 5% real (1,000 pairs)
   ↓
95% synthetic heuristic
   ↓
XGBoost accuracy: 79%
```

2. Cache Generation (One-Time)

```
# Terminal command
python generate_real_data.py 5000

# What it does:
For each of 5000 jobs:
    For each of 4 prompts:
        hash = md5(job + prompt)
        if cache exists:
            skip (already have it)
        else:
            result = ollama.chat(job, prompt)
            save to cache/{hash}.json
```

Progress output:

```
[CACHE] Processing 20,000 job-prompt pairs...
[CACHE] 1000/20000 (5%) - ETA: 90 min
[CACHE] 5000/20000 (25%) - ETA: 60 min
[CACHE] 10000/20000 (50%) - ETA: 45 min
[CACHE] 15000/20000 (75%) - ETA: 30 min
[CACHE] 20000/20000 (100%) - DONE!
[CACHE] Generated 19,000 new cache files
[CACHE] Reused 1,000 existing cache files
```

3. Training With Full Cache (Fast)

```
# predictors/mlbp.py
GROUNDING_RATIO = 1.0  # ← SET THIS

# Then run:
python main.py
```

What happens internally:

```
create_training_data():
    For job in jobs (5000):
        For prompt in prompts (4):
            ✓ Call query_gemini(job, prompt)
            ✓ Instantly returns from cache
            ✓ label = 1 if LLM correct else 0
            ✓ X.append(features)
            ✓ y.append(label)

    return X (20000×7), y (20000×1)
    ↓
train_predictor(X, y):
    XGBoost fits on 20,000 REAL samples
    ↓
    Accuracy: 85%+ (expected)
```

4. Prediction & Optimization (Unchanged)

```
predict_probabilities():
    For each job-prompt pair:
        P(success | job, prompt) from trained XGBoost

    Returns: 5000×4 probability matrix
    ↓
Optimizers (NSGA-II, SPEA2, Random):
    Use probabilities to find optimal schedules
    ↓
Pareto Front → Cost-Accuracy Tradeoff
```

Key Insights

Why Cache Matters

Without Cache (API calls every run):

```
Run #1: main.py → 1,000 API calls → 60 sec wait
Run #2: main.py → 1,000 API calls → 60 sec wait
```

```
Run #3: main.py → 1,000 API calls → 60 sec wait
Total: 3,000 calls, 180 seconds
```

With Cache (one-time generation):

```
Generation: generate_real_data.py → 20,000 calls → 120 min (ONE TIME)
Run #1: main.py → 0 API calls → 10 sec (cache hit)
Run #2: main.py → 0 API calls → 10 sec (cache hit)
Run #3: main.py → 0 API calls → 10 sec (cache hit)
Total: 20,000 calls once, then instant reruns
```

Why 100% Real Data Improves Accuracy

Current Heuristic (95% of training data):

```
# Oversimplified pattern
if is_anomaly:
    success = 0.65 + 0.20 * prompt_tokens/100
else:
    success = 0.80 + 0.10 * prompt_tokens/100
```

❌ Doesn't capture real LLM failure modes
❌ Doesn't capture prompt-specific quirks
❌ Doesn't capture job-specific edge cases

Real LLM Data (100% of training data):

```
# Actual Ollama gemma3 responses
Job: "PacketResponder terminating" + simple → Correct ✓
Job: "PacketResponder terminating" + fewshot_3 → Correct ✓
Job: "DataNode shutdown" + simple → Wrong ✗
Job: "DataNode shutdown" + fewshot_3 → Correct ✓
```

✅ Learns real LLM strengths/weaknesses
✅ Learns which prompts help which job types
✅ Learns actual failure patterns

Summary

Question	Answer
What are 20,000 calls?	5,000 jobs × 4 prompts = 20,000 LLM classifications
Where are they saved?	`results/ollama_cache/{hash}.json` (20,000 JSON files)

Question	Answer
How to generate?	`python generate_real_data.py 5000` (one-time, 120 min)
How to use?	Set `GROUNDING_RATIO=1.0`, run `main.py` (10 sec)
Why it helps?	XGBoost learns from 100% real LLM behavior → 85%+ accuracy
Current state?	Only 5% cached (1,000 files), 95% synthetic
Target state?	100% cached (20,000 files), 0% synthetic

Next Steps:

1. Run `python generate_real_data.py 5000` (wait 120 min)
2. Check `results/ollama_cache/` has ~20,000 JSON files
3. Edit `predictors/mlbp.py`: Change `GROUNDING_RATIO = 1.0`
4. Run `python main.py` (should complete in 10 sec)
5. Verify accuracy improved from 79% → 85%+