The 3 Biggest Challenges in Fine-Tuning Large Language Models (LLMs) in 2025

#A Frontend Developer's Perspective

Submitted by: TaanVeer **University:** Hajee Mohammad Danesh Science & Technology University, Dinajpur, Bangladesh **Date:**01 November 2025

Table of contents

ABSTRACT

This paper examines the three primary challenges frontend developers face when fine-tuning Large Language Models (LLMs) in 2025: the **data quality paradox**, the **cost-performance tightrope**, and the **black box problem**. As LLMs become increasingly central to web applications, frontend developers must reconcile usability, transparency, and performance while collaborating closely with ML, backend, and operations teams.

Findings indicate that teams should first attempt **prompt engineering** and **Retrieval-Augmented Generation (RAG)** before investing in fine-tuning. When fine-tuning is warranted, a methodical approach with proper tooling — including validation UIs, observability, and human-in-the-loop workflows — substantially reduces risk and cost. This edition adds practical code examples, observability guidance, safety/bias mitigation patterns, and cross-disciplinary collaboration strategies.

1. Introduction

The adoption of LLMs within production web systems has shifted from experimental to foundational. In 2025, frontend developers are not just UI implementers; they are active participants in applying machine learning to real user experiences.

This convergence requires new workflows that bridge frontend engineering, MLOps, backend systems, and data science. Effective LLM integration depends on shared responsibility, transparent tooling, and iterative collaboration across teams.

This paper analyzes the three most pressing challenges for frontend teams involved in LLM fine-tuning and proposes practical solutions supported by code examples and best practices.

Executive Summary

Challenge	Core Problem	Frontend Role	Suggested Solution
Data Quality Paradox	Biased, inconsistent, or poorly labeled training data	Surface & validate dataset examples; implement UI safeguards	Data validator UIs, human-in-the-loop review, dataset provenance tools
Cost-Performance Tightrope	High compute and serving costs versus latency/UX needs	Monitor and visualize costs; implement caching strategies	Cost dashboards, caching RAG results, distillation choices
Black Box Problem	Opaque model reasoning complicates debugging and trust	Log interactions, enable model comparisons, present explanations	Observability stacks, model comparison UIs, explainability tools

2. Challenge One — The Data Quality Paradox

The core issue

Fine-tuning requires high-quality, domain-appropriate data. Frontend engineers typically do not own dataset creation yet must manage how outputs appear to users — creating an accountability gap when model outputs are biased or inconsistent.

Key considerations

- **Bias and representation:** Linguistic, demographic, and topical skews can degrade UX and fairness.
- **Data provenance:** Lack of traceability for training examples complicates remediation.
- **Human oversight:** Human review remains necessary for sensitive domains.

Practical solution: Data validation interfaces

Build intuitive UIs that allow teams to review, flag, and batch-correct examples. Such tools accelerate detection of anomalies and facilitate human-in-the-loop workflows prior to fine-tuning.

Code example — React: *DataValidator*

```
function DataValidator({ examples, onFlag }) {
  const [flagged, setFlagged] = useState(new Set());

  const flagAsOutlier = (index) => {
    const newSet = new Set(flagged);
    newSet.add(index);
    setFlagged(newSet);
    onFlag && onFlag(index);
  };

  return (
    <div className="data-validator">
      {examples.map((ex, i) => (
        <div key={i} className={`example-card ${flagged.has(i) ? 'flagged' : ''}`}>
          <div className="example-header">
            <span>Example #{i + 1}</span>
            <button onClick={() => flagAsOutlier(i)} className="flag-btn">
              {flagged.has(i) ? '✓ Flagged' : 'Flag Outlier'}
            </button>
          </div>
          <pre className="example-content">{JSON.stringify(ex, null, 2)}</pre>
        </div>
      ))}
    </div>
  );
}
```

Note: Complement front-end validators with backend checks (schema validation, automatic bias detectors) and provenance logging so flagged items can be traced to source datasets.

3. Challenge Two — The Cost-Performance Tightrope

The core issue

Serving and fine-tuning LLMs is expensive. Frontend decisions (e.g., model choice or RAG frequency) directly affect operating costs and user experience.

Best practice: Cost observability

Implement dashboards that surface monthly/daily spend, cost per endpoint, and usage trends — enabling product teams to make budget-aware tradeoffs.

Code example — React: *CostDashboard* (partial)

```
import { useEffect, useState } from 'react';

function CostDashboard() {
  const [costData, setCostData] = useState({ monthly: 0, daily: 0, byEndpoint: {} });
  const [trend, setTrend] = useState('stable');

  useEffect(() => {
    const fetchCostData = async () => {
      try {
        const response = await fetch('/api/llm/cost');
        const data = await response.json();
        setCostData(data);

        if (data.monthly > (data.lastMonth ?? 0) * 1.2) setTrend('increasing');
        else if (data.monthly < (data.lastMonth ?? 0) * 0.8) setTrend('decreasing');
        else setTrend('stable');
      } catch (error) {
        console.error('Failed to fetch cost data:', error);
      }
    };

    fetchCostData();
    const interval = setInterval(fetchCostData, 300000);
    return () => clearInterval(interval);
  }, []);
  // Render omitted for brevity
}
```

Operational suggestions

- Cache RAG results to reduce repeated inferences.
- Use model distillation / quantization for cheaper latency-sensitive endpoints.
- Define escalation paths for unexpected cost spikes.

4. Challenge Three — The Black Box Problem

The core issue

LLM behavior is frequently opaque; understanding root causes for a specific output requires careful logging, correlation, and sometimes specialized explainability tools.

Observability & logging strategy

Log prompts, outputs, relevant metadata (model id/version, tokens, user/session context), and any intermediate retrievals (for RAG). Use correlation IDs and integrate with observability platforms (e.g., OpenTelemetry, Grafana).

Code example — React Hook: *useLLMLogger*

```
import { useCallback, useContext } from 'react';
import { UserContext } from './UserContext';

function useLLMLogger() {
  const user = useContext(UserContext);

  const logLLMInteraction = useCallback(async (prompt, output, metadata = {}) => {
    const logEntry = {
      prompt,
      output,
      timestamp: new Date().toISOString(),
      user: user?.id || 'anonymous',
      ...metadata
    };

    try {
      await fetch('/api/log/llm-interaction', {
        method: 'POST',
        body: JSON.stringify(logEntry),
        headers: { 'Content-Type': 'application/json' }
      });
    } catch (error) {
      console.error('Failed to log LLM interaction:', error);
    }
  }, [user]);

  return logLLMInteraction;
}
```

Model comparison UI

Provide side-by-side output comparison across models/versions to make decisions about selection and tuning more transparent.

Code example — React: *ModelComparison* (partial)

```
function ModelComparison({ input, models, onSelect }) {
  const [outputs, setOutputs] = useState({});
  const [loading, setLoading] = useState({});

  const generateOutputs = useCallback(async () => {
    const newOutputs = {};
    for (const model of models) {
      setLoading(prev => ({ ...prev, [model.id]: true }));
      try {
        const res = await fetch('/api/generate', {
          method: 'POST',
          body: JSON.stringify({ prompt: input, model: model.id, version: model.version })
        });
        const data = await res.json();
        newOutputs[model.id] = data.output;
      } catch (err) {
        newOutputs[model.id] = `Error: ${err.message}`;
      } finally {
        setLoading(prev => ({ ...prev, [model.id]: false }));
      }
      setOutputs(prev => ({ ...prev, ...newOutputs }));
    }
  }, [input, models]);

  useEffect(() => {
    if (input && input.length > 5) generateOutputs();
  }, [input, generateOutputs]);
}
```

5. Safety & Bias Handling

Integrate bias reporting and editable previews directly into UIs to increase transparency and create channels for corrective feedback. Flagged items should be routed to annotation queues or ML teams for triage and potential dataset updates.

Code example — React: *EditableOutput*

```
function EditableOutput({ value, onChange, onReport }) {
  const [isEditing, setIsEditing] = useState(false);
  const [editedValue, setEditedValue] = useState(value);

  const handleSave = () => {
    onChange && onChange(editedValue);
    setIsEditing(false);
  };

  const handleReportBias = () => {
    onReport && onReport({ original: value, edited: editedValue, reason: 'bias', timestamp: new Date().toISOString() });
  };

  return (
    <div className="editable-output">
      <div className="output-banner">⚠️ AI-generated content — review before use.</div>
      {isEditing ? (
        <div className="editing-mode">
          <textarea value={editedValue} onChange={(e) => setEditedValue(e.target.value)} rows="4" />
          <div className="editor-actions">
            <button onClick={handleSave} className="btn">Save Changes</button>
            <button onClick={() => setIsEditing(false)} className="btn" style="background:#64748b">Cancel</button>
          </div>
        </div>
      ) : (
        <div className="view-mode">
          <div className="output-content">{value}</div>
          <div className="output-actions">
            <button onClick={() => setIsEditing(true)} className="btn">Edit Output</button>
            <button onClick={handleReportBias} className="btn" style="background:var(--danger)">Report Bias/Error</button>
          </div>
        </div>
      )}
    </div>
  );
}
```

Important: Ensure reports are privacy-preserving (avoid logging personal data), and set SLAs for triage and remediation.

6. Teamwork & MLOps Integration

Cross-functional squads combining frontend, backend, ML, and ops are essential. Shared tools (dashboards, annotation interfaces, test suites) and regular review cycles help close feedback loops for model improvement and UX alignment.

- Define shared metrics: accuracy, hallucination rate, cost per request, time-to-fix for flagged outputs.
- Adopt common observability and provenance tooling across environments.
- Practice iterative rollouts with canary testing and user feedback collection.

7. Conclusion

Fine-tuning LLMs in 2025 presents technical and organizational challenges for frontend developers. Nonetheless, by adopting proactive tooling, embedding safety in design, and fostering collaborative cultures, teams can leverage fine-tuning effectively while mitigating operational, ethical, and cost risks.

Takeaways:

1. **Proactive tooling:** validation UIs, cost dashboards, and comprehensive logs.
2. **Safety in design:** editable outputs, reporting, and human review workflows.
3. **Collaborative culture:** cross-team squads and shared instrumentation.

References

1. TanStack Query Documentation — tanstack.com/query
2. Zod Validation Library — github.com/colinhacks/zod
3. Redux Toolkit — redux-toolkit.js.org
4. Web Streams API — developer.mozilla.org
5. React Error Boundaries — reactjs.org
6. OpenTelemetry — opentelemetry.io
7. MLOps resources — ml-ops.org
8. Buolamwini, J., & Gebru, T. (2018). Fairness and Bias study — proceedings.mlr.press