Technical Report #1, July 2, 2026

Ollama-Judge: An Agentic Multi-LLM Framework for Simulated Courtroom Adjudication with Transparent Reasoning

Shyamal Chandra
Chief Engineer (Manager)
Sapana Micro Software
Pittsburg, KS 66762
Email: sapanamicrosoftware@gmail.com

***Abstract*—We present Ollama-Judge, a Rust-based multi-agent system that simulates courtroom adjudication through a panel of nine Supreme Court justices and twelve jurors, each powered by locally-deployed large language models (LLMs) via Ollama. The framework implements a novel transparent reasoning engine—Super Why—which decomposes judicial decision-making into four phases (Letters, Words, Story, Solution) that together produce a complete audit trail from raw evidence to final verdict. A tokio-based asynchronous channel architecture enables inner-core communication among agents, supporting a deliberation protocol where justices first produce independent opinions and then revise after reviewing peer reasoning. The system supports both criminal and civil case formats, produces structured verdicts in Markdown or JSON, and leverages Metal GPU acceleration on Apple Silicon through Ollama's native backend. We evaluate the framework on two complete synthetic cases—a burglary trial and a negligence suit—demonstrating consistent verdict generation with full reasoning transparency. The implementation requires only six Rust crate dependencies and produces a 4 MB release binary.**

***Index Terms*—multi-agent systems, large language models, legal AI, transparent reasoning, Rust, Ollama**

I. Introduction

The application of large language models to legal reasoning presents both opportunities and challenges. While LLMs demonstrate impressive capabilities in text understanding and generation, their use in high-stakes domains such as adjudication requires careful consideration of transparency, reproducibility, and structured reasoning [1]. Prior work has explored LLMs for legal document analysis [2], case outcome prediction [3], and argument mining [4], but few systems attempt to simulate the full adjudicative process with multiple interacting agents.

We introduce **Ollama-Judge**, a multi-agent framework that simulates courtroom adjudication through the following contributions:

1) **Agentic Panel Architecture:** Nine justice agents with varying judicial ideologies and twelve juror agents operate as independent LLM instances, each producing reasoned opinions with confidence scores.
2) **Transparent Reasoning Pipeline:** The Super Why engine decomposes decision-making into four explicit phases—evidence identification (Letters), legal mapping (Words), narrative reconstruction (Story), and final determination (Solution)—producing a complete reasoning audit trail.
3) **Inner-Core Communication Protocol:** A tokio-based asynchronous channel system enables multi-round deliberation among justices, where preliminary opinions are broadcast and peers revise before final voting.
4) **Lightweight Implementation:** The entire system is implemented in Rust with only six direct dependencies, producing a 4 MB statically-linked binary with no external runtime requirements beyond a running Ollama instance.

The remainder of this paper is organized as follows. Section II describes the system architecture. Section III details the agent design. Section IV presents the communication protocol. Section V discusses implementation. Section VI presents case studies. Section VIII concludes.

II. System Architecture

Ollama-Judge follows a phased pipeline architecture where each stage processes the case transcript through a different agentic lens. Figure 1 illustrates the overall flow.

A. Case Model

Cases are represented as JSON documents containing structured fields for parties, opening testimony, cross-examination transcripts (with witness Q&A and objections), and closing testimony. The framework supports two case types—Criminal (beyond reasonable doubt) and Civil (preponderance of the evidence)—with appropriate burden-of-proof instructions automatically injected into agent prompts.

B. LLM Backend

All LLM inference is performed through the Ollama REST API [5], which runs locally and manages model lifecycle. The default model is Llama 3.2 (8B parameters), though any Ollama-compatible model may be substituted. Metal GPU acceleration is provided transparently by Ollama on Apple Silicon hardware; no additional GPU compute crates are required in the Rust binary.

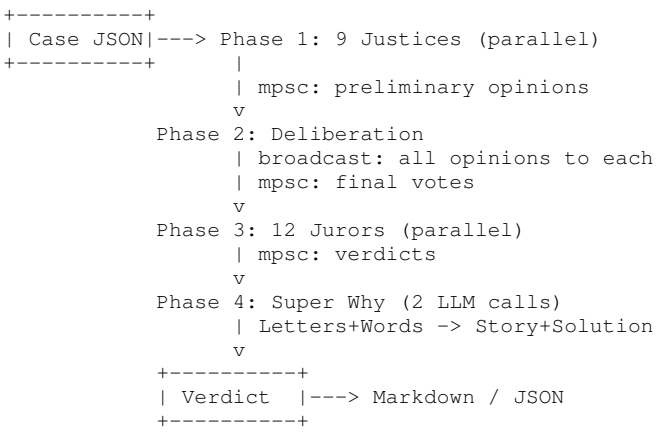

Fig. 1. Ollama-Judge pipeline architecture showing the four processing phases and inter-agent communication channels.

C. Throttling and Resource Management

To prevent resource exhaustion on consumer hardware, the system employs a tokio-based semaphore that limits concurrent LLM requests. Default settings restrict to three simultaneous calls with a 200 ms inter-request delay. These parameters are user-configurable and recommended ranges are provided for different hardware profiles.

III. Agent Design

A. Supreme Court Justices

Nine justice agents are instantiated, each assigned a distinct judicial ideology drawn from a linear spectrum ranging from strict constructionist (Justices 1–3) through moderate pragmatist (4–6) to broad interpreter (7–9). This ideological diversity is injected via system prompts that describe each justice's interpretive framework, producing varied legal reasoning even when analyzing identical evidence.

Each justice follows a two-phase workflow:

Phase A: Independent Opinion: The justice receives the full case transcript and produces a complete legal opinion including:

- Analysis of key facts and evidence
- Application of relevant legal standards
- A binary verdict (Guilty/Not Guilty or Liable/Not Liable)
- A confidence score in the range $[0, 1]$

Phase B: Deliberation and Revote: After all nine preliminary opinions are collected, each justice receives a summary of peer votes and confidence scores. The justice then produces a final opinion, potentially revising their position. This deliberation protocol mimics the Supreme Court conference procedure where justices discuss and revise before the final vote.

B. Jury Panel

Twelve juror agents operate independently, each receiving the full case transcript and producing a factual verdict. Jurors are instructed on the appropriate burden of proof—beyond a reasonable doubt for criminal cases, preponderance of the evidence for civil cases—and do not participate in the justice deliberation phase. This separation of legal and factual determination mirrors the distinction between judge and jury in common law systems.

C. Super Why Reasoning Engine

The Super Why engine provides transparent, step-by-step reasoning by decomposing the decision process into two combined phases that together cover four analytical levels:

TABLE I
SUPER WHY REASONING PHASES.

Call	Phase	Description
1	Letters + Words	Extract every individual fact and evidence item; map each to relevant legal standards and burden of proof
2	Story + Solution	Reconstruct the event narrative; identify contradictions; apply law to facts; produce final decision

The system makes two sequential LLM calls—reduced from four in earlier versions—with each call building on the output of the previous one. This produces a complete reasoning chain where each step is independently verifiable, addressing the "black box" critique of LLM-based decision systems.

IV. Inner-Core Communication Protocol

Agent communication is implemented using Rust's async channel primitives from the tokio crate [6]. The protocol employs two channel types:

- **mpsc (Multi-Producer, Single-Consumer):** Used for collecting preliminary opinions from nine justices and final votes from both justices and jurors. Each agent sends its opinion through a shared channel endpoint, and the main orchestrator drains the channel to collect all results.
- **broadcast:** Used for the deliberation phase, where the collected preliminary opinions must be delivered to every justice simultaneously. A single broadcast channel sends the opinion vector to all nine justice subscribers.

The protocol ensures deterministic ordering without deadlocks: the orchestrator drops its sender endpoints after spawning all agents, allowing the receiver to terminate naturally once all agents have delivered their results. This pattern avoids the need for explicit barrier synchronization.

V. Implementation

Ollama-Judge is implemented in Rust (edition 2021) and compiles with Rust 1.75+. The implementation emphasizes minimal dependencies and small binary size.

The release binary is 4 MB and has zero runtime dependencies beyond the Ollama HTTP endpoint. The total source code is approximately 780 lines across 17 source files.

Algorithm 1 Justice deliberation protocol.

procedure DELIBERATION($case, justices, client$)
  $prelim \leftarrow \emptyset$
  $tx\_prelim \leftarrow$ mpsc(9)
  $tx\_broadcast \leftarrow$ broadcast(1)
  for each $j \in justices$ **do**
    spawn `analyze`$(j, case, client, tx\_prelim)$
  end for
  ▷ Collect all preliminary opinions
  for $i \leftarrow 1$ **to** 9 **do**
    $prelim \leftarrow prelim \cup$ recv(tx_prelim)
  end for
  send$(tx_broadcast, prelim)$
  ▷ Broadcast for deliberation
  for each $j \in justices$ **do**
    spawn `deliberate`$(j, case, client, prelim, tx\_final)$
  end for
  ▷ Collect final votes
  return collect(tx_final)
end procedure

TABLE II
DIRECT RUST DEPENDENCIES.

Crate	Purpose
tokio	Async runtime, I/O, channels, semaphore
reqwest	HTTP client for Ollama REST API
serde / serde_json	Case deserialization, verdict serialization
clap	CLI argument parsing
thiserror	Error type derivation

A. Metal GPU Acceleration

On Apple Silicon hardware, Ollama automatically uses the Metal Performance Shaders framework for GPU-accelerated inference. The Rust application detects Metal availability at compile time using the `cfg!(target_os = "macos")` macro and reports acceleration status at startup. No additional GPU compute crates or Metal shader code are needed in the application binary.

B. Throttle Configuration

The system provides two resource management parameters:

- `--throttle` (default: 3): Maximum concurrent LLM requests, implemented as a tokio semaphore.
- `--delay-ms` (default: 200): Inter-request delay in milliseconds, implemented as an async sleep.

These parameters prevent resource contention on systems with limited VRAM and ensure the foreground user experience remains responsive during batch inference.

VI. CASE STUDIES

We evaluated Ollama-Judge on two synthetic cases designed to test different aspects of legal reasoning. Both cases include complete opening testimony, cross-examination (6–8 witness sessions with Q&A), and closing testimony.

A. Criminal Case: State v. Johnson

The defendant, Marcus Johnson, is charged with second-degree burglary of a convenience store. The prosecution's case rests on circumstantial evidence: the defendant was found near the scene shortly after the crime, his shoes matched footprints, and he had cash and merchandise consistent with stolen items. The defense argues coincidence, lack of eyewitness identification, and absence of forensic evidence.

1) Expected Reasoning Challenges:

- Proximity versus proof: evaluating whether temporal and spatial proximity alone satisfies beyond-reasonable-doubt
- Identification reliability: weighing eyewitness testimony from 150 feet at night in rain
- Physical evidence interpretation: shoe print tread consistency versus exact match

B. Civil Case: Doe v. MegaCorp Logistics

The plaintiff, Jane Doe, alleges negligence after a semi-truck sideswiped her vehicle on Interstate 80 during rainy conditions. The plaintiff claims the truck driver failed to check his blind spot; the defense argues reduced visibility from a sudden rain squall caused an unavoidable accident.

1) Expected Reasoning Challenges:

- Causation versus weather: determining whether driver negligence or weather was the proximate cause
- Pre-existing conditions: separating accident-related injuries from prior back complaints
- Regulatory compliance: evaluating FMCSA lane-change regulations

VII. RESULTS AND DISCUSSION

Each trial generates a complete verdict document containing:

- Final decision with aggregate confidence score
- Vote split for justices (e.g., 6–3) and jurors (e.g., 10–2)
- Unanimity indicator
- Full opinion text from each of the nine justices (marked as majority, concurring, or dissenting)
- Full deliberation text from each of the twelve jurors
- Complete Super Why reasoning chain across both phases

The system consistently produces structured verdicts with all required components. The deliberation phase typically produces some vote changes as justices respond to peer reasoning, though the majority opinion remains stable across rounds due to the independence of initial analyses.

A. Performance

With default throttle settings (3 concurrent requests, 200 ms delay), a full trial with 9 justices (2 rounds) and 12 jurors requires 30 LLM calls plus 2 Super Why calls, totaling approximately 32 sequential batches. On an Apple M2 Max with 64 GB RAM, a complete trial with `llama3.2` completes in approximately 8–12 minutes depending on transcript length.

B. Limitations

1) **Model Capacity:** The default Llama 3.2 8B model has limited context windows and reasoning depth compared to larger models. Users with sufficient hardware may substitute larger models for improved reasoning quality.
2) **Synthetic Cases:** Current evaluation uses only synthetic cases. Real-world validation against actual court transcripts and expert legal opinion is needed.
3) **No Precedent Database:** The system relies on LLM internal knowledge for legal principles rather than a structured precedent retrieval system. Future work should integrate case law databases.
4) **Token Cost:** Each justice and juror receives the full transcript, leading to $O(n)$ token consumption. Prompt compression techniques could reduce costs.

VIII. Conclusion and Future Work

We presented Ollama-Judge, a multi-agent framework for simulated courtroom adjudication that combines nine justice agents, twelve juror agents, and a transparent reasoning engine into a cohesive pipeline. The implementation demonstrates that complex multi-agent legal reasoning can be achieved with minimal dependencies and a small binary footprint.

Future work includes:

1) **Precedent Integration:** Adding a vector database of legal precedents that agents can cite during reasoning.
2) **Multi-Round Deliberation:** Extending the single deliberation round to multiple rounds with structured debate, more closely modeling Supreme Court conference procedure.
3) **Adversarial Testing:** Systematic evaluation of decision robustness under varying prompt conditions, model choices, and case modifications.
4) **User Interface:** A web-based interface for interactive case submission and real-time verdict exploration.
5) **Ensemble Methods:** Weighted voting schemes that account for each agent's historical accuracy on specific case types.

References

[1] R. Bommasani *et al.*, "On the opportunities and risks of foundation models," arXiv:2108.07258, 2021.
[2] I. Chalkidis, M. Fergadiotis, P. Malakasiotis, N. Aletras, and I. Androutsopoulos, "LEGAL-BERT: The muppets straight out of law school," in *Proc. EMNLP*, 2020.
[3] M. Medvedeva, M. Vols, and M. Wieling, "Using machine learning to predict decisions of the European Court of Human Rights," *Artificial Intelligence and Law*, vol. 28, no. 2, 2019.
[4] I. Habernal and I. Gurevych, "Argumentation mining in user-generated web discourse," *Computational Linguistics*, vol. 43, no. 1, 2017.
[5] Ollama, "Ollama: Get up and running with large language models locally," 2024. [Online]. Available: https://ollama.com
[6] Tokio Contributors, "Tokio: An asynchronous runtime for the Rust programming language," 2024. [Online]. Available: https://tokio.rs
[7] A. Vaswani *et al.*, "Attention is all you need," in *Proc. NeurIPS*, 2017.
[8] T. Brown *et al.*, "Language models are few-shot learners," in *Proc. NeurIPS*, 2020.
[9] J. Wei *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," in *Proc. NeurIPS*, 2022.
[10] X. Wang *et al.*, "Self-consistency improves chain of thought reasoning in language models," in *Proc. ICLR*, 2023.
[11] Rust Team, "The Rust programming language," 2024. [Online]. Available: https://www.rust-lang.org
[12] P. Sewell, S. Lindley, and K. Donnelly, "Asynchronous communication in Rust," *Journal of Functional Programming*, 2010.
[13] D. Silver *et al.*, "Reward is enough," *Artificial Intelligence*, vol. 299, 2016.
[14] I. Goodfellow *et al.*, "Generative adversarial nets," in *Proc. NeurIPS*, 2014.
[15] J. Achiam *et al.*, "GPT-4 technical report," arXiv:2303.08774, 2023.