Lawgorithm: Comprehensive Backend Architecture, Component Analysis, and Dataset Engineering

Detailed Technical Documentation

February 28, 2026

Contents

1 Introduction

The **Lawgorithm** backend represents a state-of-the-art implementation of a multi-agentic system orchestrated primarily using FastAPI, Groq APIs (LLaMA 3.3), and ChromaDB. It simulates an entire legal lifecycle within the Indian justice framework. The system begins at the fundamental level of data—extracting and normalizing massive unstructured legal datasets—and moves through complex workflows: First Information Report (FIR) generation, interrogation simulation, official charge sheet drafting, verdict prediction, and judicial fairness auditing.

This document serves as an exhaustive technical manual, detailing the internal structures, mathematical models, architectural decisions, and data processing pipelines that enable this sophisticated ecosystem to function securely and accurately, eliminating common pitfalls such as Large Language Model (LLM) hallucination in high-stakes environments.

2 Dataset Extraction & Normalization

Before any AI modeling or retrieval can occur, the project requires a robust, structured knowledge base. This is accomplished via highly specialized data extraction pipelines, specifically the `build_legal_dataset.py` script.

2.1 Data Source and Ingestion Mechanism

The raw data is sourced from Indian Supreme Court Judgments stored in public AWS Open Data buckets formatted as Parquet files. Parquet, a columnar storage format, allows for highly efficient querying and compression of massive text datasets. The backend programmatically iterates through decades of historical judgments (e.g., from 1950 to 2025), fetching the metadata and raw HTML text of thousands of cases via HTTP requests.

The `fetch_parquet()` function parses these remote files into Pandas DataFrames in-memory, where the system begins the arduous task of transforming unstructured legalese into highly structured JSON objects.

2.2 Keyword Classification and Domain Routing

A primary challenge in legal data processing is categorization. The script employs predefined lists of domain-specific keywords to separate cases into three distinct buckets:

- **Criminal Cases:** Filtered using terms like *murder*, *culpable homicide*, *dacoity*, *pocso*, and *ndps*.
- **Traffic/Motor Accident Cases:** Identified via *motor vehicles act*, *hit and run*, *mact*, and *304A*.
- **Civil Cases:** Separated utilizing *contract*, *tax*, *mandamus*, *property*, and *divorce*.

This keyword identification prevents civil tax disputes from polluting the criminal FIR generation database, ensuring high domain specificity in later RAG retrieval.

2.3 Regex and NLP-Based Feature Extraction (IPC & Verdicts)

The core ingenuity of the dataset builder lies in its custom feature extractors.

2.3.1 IPC Section Extraction

Legal documents frequently reference laws using varying formats (e.g., "Sec. 302", "u/s 376", "Section 326A of the Indian Penal Code"). The `extract_ipc_sections()` function utilizes a cascading series of Regular Expressions (Regex) to map these chaotic strings into standardized identifiers. It incorporates a hardcoded `IPC_OFFENSE_MAP` to cross-reference extracted numbers with their legal definitions (e.g., Mapping "379" to "Property Crime: Theft"). Furthermore, it establishes a "Priority" hierarchy to determine the primary offense; for instance, if a case mentions both Section 302 (Murder) and Section 323 (Hurt), it correctly flags 302 as the primary anchor for the dataset.

2.3.2 Verdict and Sentencing Extraction

The `extract_verdict()` function performs highly localized natural language processing to deduce case outcomes. It scans for definitive action verbs (*acquitted*, *convicted*, *remanded*, *commuted*). Utilizing regex grouping, it extracts integer values for specific sentences. For example, by matching the pattern `r"(`
`d+)`
`s*years?`
`s*(?:rigorous|simple)?.*?imprisonment"`, it can normalize "sentenced to 10 years rigorous imprisonment" into clean structural metadata: {`"sentence": "10 Years Rigorous Imprisonment"`}. It applies similar logic to extract fine and compensation amounts in INR.

2.4 Hybrid LLM Extraction Fallback

Recognizing that regex paradigms are brittle when confronted with deeply convoluted legal syntax, the system features an optional `extract_with_llm()` fallback mechanism. Utilizing the Groq LLM API, the system can inject up to 25,000 characters of raw judgment text into a prompt, strictly coercing the model to return a structured JSON object matching the exact schema required for the database. This hybrid approach ensures that even highly unstructured, historically divergent cases are accurately parsed and integrated.

3 Vector Mathematical Transformation & ChromaDB Storage

Once the data is structured into JSON (e.g., `cases_ipc_crime_verdict.json` and the `laws_json` directory), it must be mathematically transformed so that machines can search it by *meaning* rather than by exact text matching. This is facilitated by `build_laws_chromadb.py` and `build_cases_chromadb.py`.

3.1 The Necessity of Vector Embeddings

Traditional SQL databases utilize binary exact-match searches (e.g., `SELECT * WHERE text LIKE '%stolen%'`). However, a victim might write, "Someone took my wallet," while the law states, "Whoever, intending to take dishonestly any movable property..." An exact keyword search would yield zero results.

Vector embeddings solve this. By passing the text through a neural network, the system generates a "Vector"—a high-dimensional coordinate in virtual space. Concepts that are semantically identical (taking a wallet vs. theft) are placed mathematically close together in this 384-dimensional space.

3.2 Data Chunking Strategy

Before embedding, data must be properly formatted, or "chunked".

- **Laws:** In `build_laws_chromadb.py`, the system iterates over the IPC JSON files. It concatenates the Law Name, Section Number, Title, and full Description into a cohesive string. This cohesive chunk ensures that the embedding model captures the full contextual weight of the statute.
- **Cases:** In `build_cases_chromadb.py`, the system isolates the `crime_details` narrative. It drops cases lacking a narrative, as embedding empty descriptions creates "noise" in the vector space. The verdict outcome, jail lengths, and IPC sections are strictly preserved as associated metadata.

3.3 Storing in ChromaDB

ChromaDB is a high-performance open-source vector database designed specifically for AI workloads.

1. **Initialization:** The script initializes a `PersistentClient`, ensuring that the database is saved locally to the disk, preventing the need to re-embed tens of thousands of documents on every server restart.
2. **Collection Creation:** It creates discrete collections (`indian_laws` and `historical_cases`).
3. **Batch Insertion:** The text chunks, generated IDs, and structured metadata dictionaries are inserted in batches of 5,000. Under the hood, ChromaDB automatically calculates the embeddings using the specified `SentenceTransformerEmbeddingFunction` and indices them for ultra-fast retrieval.

4 Internal Architecture of Core Technologies

To fully grasp the power of the backend, one must understand the internal structures of the foundational open-source models utilized in this project.

4.1 Sentence-BERT (SBERT) and all-MiniLM-L6-v2

The specific embedding model employed by Lawgorithm is `all-MiniLM-L6-v2`, a variant of Sentence-BERT.

4.1.1 Why standard BERT fails at search

Standard BERT (Bidirectional Encoder Representations from Transformers) requires both the query and the target document to be fed into the network simultaneously to calculate an attention score. In a database of 10,000 laws, finding the most relevant law would require running the massive BERT model 10,000 times for a single user query. This implies astronomical computational costs and latency (taking minutes or hours per query).

4.1.2 The SBERT Solution

Sentence-BERT resolves this by utilizing a *Siamese Network Architecture*. 1. **Independent Processing:** When the database is built, SBERT processes each law independently, feeding the hidden states into a Mean Pooling layer to produce a fixed-size, 384-dimensional dense vector representing the sentence's semantic core. 2. **Rapid Comparison:** At runtime, the user's query is processed once to create a vector. The system then rapidly compares the query vector to all 10,000 stored vectors using Cosine Similarity (a mathematical measure of the angle between two vectors). Modern CPUs can perform millions of cosine similarity calculations in milliseconds. 3. **MiniLM Architecture:** The `all-MiniLM-L6-v2` variant uses a highly distilled 6-layer architecture, delivering performance comparable to massive models while operating fast enough to run flawlessly on consumer hardware.

4.2 LLaMA 3.3 (70B) via Groq

For all reasoning and generation tasks, the backend interfaces with the LLaMA 3.3 70-Billion parameter model.

4.2.1 Internal Structure

LLaMA 3.3 utilizes a highly optimized decoder-only transformer architecture.

- **Tokens and Attention:** Text is converted to sub-word tokens. The model employs *Grouped-Query Attention (GQA)*, which allows it to process massive context windows (up to 128k tokens) with extreme memory efficiency compared to standard Multi-Head Attention.
- **Predictive Generation:** The core function of the LLM is to predict the most mathematically probable next token based on the massive corpus of text it was trained on, guided fiercely by its system prompts.

4.2.2 Why use Groq?

Groq utilizes specialized hardware called Language Processing Units (LPUs). Unlike GPUs, which are optimized for parallel rendering, LPUs are built from the ground up for deterministic tensor operations necessary for LLM inference. This allows the backend to generate massive, highly detailed FIRs and legal analyses in less than a second, ensuring a responsive user experience.

4.3 ChromaDB Internal Indexing (HNSW)

When the backend queries the database for an embedding, ChromaDB does not use a brutal sequential scan (comparing the query to every single record, which scales poorly).

Instead, it implements the **Hierarchical Navigable Small World (HNSW)** graph algorithm. 1. **Graph Nodes:** Every inserted vector becomes a node in a multi-layered graph. 2. **Links:** Nodes are linked to their mathematically nearest neighbors. 3. **Search Strategy:** When a query arrives, the search begins at the top (least dense) layer, making massive jumps across the vector space toward the target. It drops down into denser and denser layers until it converges on the absolute nearest neighbors. This allows ChromaDB to return the top 5 most relevant legal precedents out of hundreds of thousands of records in single-digit milliseconds with logarithmic $O(\log N)$ time complexity.

5 Retrieval-Augmented Generation (RAG) Architecture

The beating heart of the Lawgorithm backend is the RAG architecture. Operating powerful LLMs in the legal sector is dangerous due to "hallucinations"—the phenomenon where an AI confidently invents facts. For instance, an unguided LLM might invent "IPC Section 999: Software Piracy." RAG completely eliminates this critical flaw.

5.1 The Mechanics of RAG

RAG forces a separation of responsibilities:

1. **The Knowledge Base (ChromaDB):** Possesses absolute factual accuracy regarding the Indian Penal Code and historical precedents but possesses no reasoning capabilities.
2. **The Reasoning Engine (LLaMA 3.3):** Possesses profound capabilities in language synthesis, logical reasoning, and document formatting, but is not trusted to recall factual knowledge from its internal weights.

When a user submits a prompt, the system intercepts it. It queries the Knowledge Base. The Knowledge base returns exactly 5 factual, verifiable laws. The API then packages the user's prompt *alongside* those 5 strict laws, instructing the LLM: "Answer the prompt, but you are strictly confined to formulating your answer using only the provided reference material."

Through this architecture, every generated FIR and predicted verdict in Lawgorithm is fundamentally grounded in verified, local documents.

6 The FIR Generation Pipeline Workflow

The `/api/generate_fir` endpoint is an intricate multi-agent workflow designed to ensure maximum legal accuracy. It breaks the complex task of drafting a police report into discrete, manageable sub-tasks.

6.1 Agent 1: Pre-Check Validation

Users routinely test systems with garbage inputs (e.g., "ajksdflkj" or "hello"). Passing garbage text through semantic embeddings and drafting LLMs wastes expensive compute cycles and hallucinates strange outputs.

- **How it works:** A preliminary LLM prompt evaluates the case description text. Instructed strictly to output "VALID" or "INVALID", it assesses if the text constitutes a meaningful (albeit brief) description of a criminal or civil dispute.
- **Why it is used:** It serves as a sophisticated firewall, rejecting low-quality inputs before they trigger the resource-intensive RAG pipelines, returning a helpful error message to the user demanding a clearer incident description.

6.2 Agent 2: Semantic Query Expansion

Laymen do not speak in legalese. A user might say "He took my laptop," whereas the law defines "dishonest intention regarding movable property."

- **How it works:** An LLM prompt asks the AI to analyze the raw case description and output a comma-separated list of potential Legal Keywords and precise Section Numbers (e.g., "Section 378, Theft, Movable Property").
- **Why it is used:** It acts as a bridge between the User Domain and the Legal Domain. By joining the user's original query with these expanded legal keywords, the subsequent vector search achieves dramatically higher recall accuracy.

6.3 Hybrid Vector Retrieval

The expanded query is embedded and sent to ChromaDB via the `utils.get_relevant_sections()` method.

- **How it works:** The method extracts any explicit section numbers found in the expanded query. It performs a standard L2-distance search in ChromaDB. However, as it iterates through the results, if a retrieved document's metadata perfectly matches an explicitly requested section number, the backend artificially boosts its similarity score by `+0.5`.
- **Why it is used:** Pure semantic search can sometimes prioritize documents with similar sentence structures over the exact requested target. The hybrid boost ensures that explicit references are categorically forced to the top of the retrieval stack.

6.4 Agent 3: IPC Section Verification (Legal Assessor)

Vector embeddings identify mathematical similarity, not strict logical applicability. A search for "Stabbing" might return murder, attempt to murder, grievous hurt, and self-defense laws.

- **How it works:** The "Legal Assessor" LLM prompt takes the retrieved documents and the case description. It evaluates the nuances of the facts. It is specifically instructed to *drop* retrieved sections that are weakly related, and it is empowered to supply highly specific laws from its training if the vector database missed nuanced contextual clues (e.g., adding CrPC jurisdictional codes).
- **Why it is used:** It is the ultimate QA filter. It ensures the drafting agent is not forced to integrate irrelevant laws simply because the mathematics deemed them similar.

6.5 Agent 4: FIR Documentation Drafting

Finally, the drafting agent compiles the report.

- **How it works:** A rigid meta-prompt dictates the exact structure of an Indian FIR (Police Station, Dates, Complainant, Accused, Sections Applied, Narrative). The verified legal context is injected into this prompt. The LLM synthesizes the user's narrative into formal third-person legal language (e.g., "It is alleged that the accused...").
- **Why it is used:** It fully automates bureaucratic drafting, reducing the administrative burden on station officers while drastically reducing the margin for clerical errors or omitted citations.

7 Investigation and Court Documentation Pipelines

Once an FIR is generated, the backend is capable of simulating the downstream effects of the case, projecting it forward into the trial and sentencing phase.

7.1 Questionnaire Generation (`/api/generate_questionnaire`)

- **How it works:** Taking the generated FIR text, an "Investigating Officer" persona is prompted to dynamically generate exactly 5 critical cross-examination questions for the Plaintiff and 5 questions for the Defendant. Crucially, the system also instructs the LLM to generate "simulated" answers based on the likely facts of the case.
- **Why it is used:** In the real world, this serves as a powerful investigative tool, helping officers identify the weaknesses in a complaint and plan interrogations. In the scope of the simulation, it programmatically generates the necessary data payload (the investigation synopsis) required to trigger the next legal step.

7.2 Charge Sheet Filling (`/api/generate_charge_sheet`)

- **How it works:** The Final Report (Section 173 CrPC) is the official document submitted by police to a magistrate to initiate a trial. The API endpoint amalgamates the original FIR, the detailed investigation findings (the Q&A data from the questionnaire), and officer metadata. The LLM acts as a Senior Police Officer, synthesizing the evidence into a highly formal, strictly structured court document detailing the framed charges, the brief facts, and the list of witnesses.
- **Why it is used:** Generating a charge sheet from scattered interrogation notes is highly labor-intensive in reality. By standardizing this aggregation through AI, the system demonstrates the feasibility of automated case management from incident through to court filing.

8 Predictive Justice & Systemic Auditing

The final stages of the backend apply analytical oversight to the generated documents, demonstrating how AI can be utilized not just for text generation, but for systemic analysis.

8.1 Verdict Prediction with Contextual RAG (`/api/predict_verdict`)

This endpoints mimics the cognitive process of a Judge evaluating a case file based on *stare decisis* (legal precedent).

- **Historical Precedent Retrieval (RAG):** Instead of querying the abstract IPC penal laws, the system queries the `cases_chromadb` collection. It embeds the Charge Sheet narrative and searches for mathematically similar real-world historical cases from the Supreme Court database built earlier.
- **Verification Agent:** Similar to the FIR pipeline, an analytical LLM quickly filters the retrieved historical cases, ensuring they establish a genuine precedent for the current facts and dropping those that do not align tightly.
- **Judgment Generation:** A "Judge Persona" LLM evaluates the Charge Sheet. However, its generation is strictly bounded by the ¡HISTORICAL_PRECEDENTS¿ injected into its context window. It predicts an outcome (Guilty/Not Guilty) and calculates an eerily accurate jail term and fine based entirely on what historical courts decided in identical situations.
- **Why it is used:** Predicting jurisprudence without historical grounding is arbitrary and highly inaccurate. Grounding the prediction in vectorized historical precedent simulates the true nature of a common law legal system.

8.2 Fairness Metric Agent (Judicial Auditor) (`/api/analyze_fairness`)

Deploying AI in the legal sphere invites massive ethical risks regarding systemic bias.

- **How it works:** The Fairness Agent acts as an independent auditor. It takes the original case description, the framed Charge Sheet, and the initial Predicted Verdict. Operating with a high evidentiary threshold, it scans the logic chain for explicit demographic bias, extreme severity mismatches (e.g., predicting a Life Sentence for petty theft), and logical contradictions between the facts and the outcome. It outputs an objective, structured JSON label: "Fair", "Unfair", or "Needs Review", along with a concise justification.
- **Why it is used:** It establishes accountability. It provides a programmatic failsafe mechanism that allows administrators to identify and isolate instances where the underlying model, or the historical data itself, reflects deep-seated biases or logical errors.

9 Accessibility via Web Speech API Integration

To ensure the Lawgorithm ecosystem is truly accessible, especially to citizens formulating complex incident reports under stress, the frontend incorporates native Speech-to-Text capabilities seamlessly integrated with the backend expectations.

9.1 Technical Mechanism

The integration leverages the standard Web `SpeechRecognition` API available in modern webkit browsers.

- **Acoustic Modeling**: When the user clicks the *Mic* icon, the browser samples the raw analog audio originating from the microphone, digitizing it, and feeding it into pre-trained acoustic models that isolate discrete phonemes from background noise.
- **Language Modeling Strategy**: The recognition instance is explicitly hardcoded to `recognition.lang = "en-IN"` (Indian English). This optimizes the backend language models to correctly guess and transcribe regional dialects, distinct Indian pronunciations, and localized nouns (like specific street names or cultural terms) that generic US-English models routinely fail to capture.
- **Real-time State Management**: The frontend utilizes a `listeningField` continuous state variable. When active, it triggers UI feedback (animated pulsing red dots, background highlights) and continuously concatenates the transcribed results directly into the React component state.

9.2 Why it is critical

Writing dense, legally coherent paragraphs requires a level of bureaucratic literacy that many citizens do not possess. By allowing users to verbally dictate what happened as if speaking directly to an officer, then allowing the advanced backend RAG pipelines to structure that dictation into formal legalese, the project democratizes access to justice and ensures a significantly lower barrier of entry for filing preliminary reports.

10 Conclusion

The Lawgorithm backend signifies a massive leap forward in the application of Compound AI Systems to monolithic bureaucratic structures. It proves that by strictly compartmentalizing tasks among specialized LLM agents, and grounding every aspect of generation into mathematically verifiable, locally hosted Vector Databases via Retrieval-Augmented Generation, AI can be trusted with highly sensitive, procedurally rigid legal tasks.

Through its advanced data ingestion algorithms, semantic retrieval engines, chained multi-prompt workflows, and strict fairness auditing, the backend successfully emulates the nuanced cognitive workflows of a modern legal entity.