Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

School of Computer Science and Engineering

Department of Computer Engineering and Technology

Third Year B. Tech. CSE (Cybersecurity and Forensics)

CSF3PM01A: Full Stack Development Laboratory

Auto AI: Mini-Project Report

Roll no.	Name of Student	PRN	Email Address	Mobile Number
46	Yashraj Narke	1032232513	yashraj.narke@mitwpu.edu.in	9284531302
48	Jayaram Patel	1032232531	jayaram.patel@mitwpu.edu.in	8265082269
52	Om Jagdeesh	1032232632	om.jagdeesh@mitwpu.edu.in	9561491121
50	Lakshya Chanchalani	1032232588	lakshya.chanchalani@mitwpu.edu.in	9717568041

Submitted to:- Prof. Sagar Apune

Sign -

Abstract

The rapid evolution of Machine Learning (ML) has led to a growing demand for efficient model development and deployment. While Automated Machine Learning (AutoML) solutions have streamlined parts of this process, they often lack the flexibility and dynamic adaptability required for diverse, real-world scenarios. A novel approach to autonomous ML model generation is presented, leveraging the advanced reasoning and code generation capabilities of Large Language Models (LLMs). The system detailed herein employs a Gemini AI-powered agent, orchestrated via LangChain, to autonomously analyze datasets, dynamically generate and execute Python code for model training, and evaluate performance. This agent interacts with a controlled execution environment through a set of specialized tools, enabling it to install dependencies, write and run training scripts, and persist model metadata. Integrated within a full-stack application, the system demonstrates a paradigm shift towards LLM-driven AutoML, offering enhanced flexibility and automation in ML workflows. The architecture, methodology, and implications of this approach are discussed, highlighting its potential to democratize ML model development while also addressing inherent challenges in robustness, debugging, and security.

Problem Statement

Despite the advancements in AutoML, many existing systems operate as black-box solutions, offering limited transparency into their decision-making processes or the underlying model architectures. They often rely on predefined templates or search spaces, which can restrict their adaptability to novel datasets or highly specialized problem requirements. Furthermore, integrating these automated solutions into broader application ecosystems can be challenging, often requiring significant custom development.

The current landscape of ML development still largely necessitates human intervention for critical tasks such as interpreting data, selecting appropriate algorithms, and debugging model failures. While LLMs can generate code, their direct application to complex, multi-step ML workflows, including iterative refinement and execution verification, remains an active area of research. A clear need exists for a more flexible, transparent, and dynamically adaptable approach to ML model generation that can seamlessly integrate into modern software architectures.

Background

The increasing complexity and diversity of data in modern applications have significantly elevated the demand for efficient and adaptive machine learning (ML) model development. Traditionally, designing, training, and deploying ML models has required substantial human expertise, iterative experimentation, and domain-specific knowledge. To mitigate this, Automated Machine Learning (AutoML) frameworks emerged, aiming to automate various stages of the ML lifecycle such as model selection, feature engineering, and hyperparameter tuning. However, conventional AutoML systems often operate within rigid pipelines and lack the flexibility to dynamically adapt to new or unstructured problem domains.

Recent advancements in Large Language Models (LLMs) have opened new possibilities for intelligent automation in ML workflows. LLMs possess powerful reasoning and code generation capabilities, enabling them to analyze datasets, write executable code, and even debug or optimize scripts in real time. Leveraging these capabilities can extend AutoML into a more autonomous, context-aware, and self-improving system.

This project builds upon that vision by developing a Gemini AI-powered agent, orchestrated using LangChain, to autonomously generate, train, and evaluate ML models. The motivation stems from the need for a more flexible and human-like automation approach that bridges the gap between rigid AutoML systems and the creative, adaptive reasoning of human ML developers. By integrating the agent into a full-stack environment with controlled execution and tool-based orchestration, this work aims to democratize access to advanced ML development, enabling non-experts to build efficient models with minimal manual intervention.

Ultimately, the project envisions a paradigm shift towards LLM-driven AutoML, where intelligent agents autonomously handle the end-to-end ML lifecycle while maintaining transparency, security, and adaptability, ushering in the next generation of automated, yet explainable, machine learning systems.

Motivation

Motivation for this project stems from the potential to harness the advanced reasoning and code generation capabilities of LLMs to create a more autonomous and adaptable ML development paradigm. By enabling an LLM to act as an intelligent agent, capable of interacting with a controlled execution environment, some limitations of traditional AutoML can be overcome. Such an agent could dynamically analyze data, write and execute custom Python scripts for model training, and iteratively refine its approach based on real-time feedback from the execution environment. The integration of this LLM-driven agent into a full-stack application, utilizing modern technologies like tRPC and Next.js, further motivates this investigation. It demonstrates how sophisticated AI capabilities can be seamlessly embedded within user-facing applications, providing a powerful and intuitive platform for ML model generation and deployment. This approach promises to democratize access to ML by abstracting away much of the underlying complexity, allowing users to focus on data and outcomes rather than intricate algorithmic details

Literature Review

The landscape of machine learning development has seen significant advancements in automation, particularly with the rise of AutoML and the increasing sophistication of LLM agents. This section contextualizes the proposed system within these existing paradigms. Traditional AutoML systems aim to automate various stages of the machine learning pipeline, including data preprocessing, feature engineering, algorithm selection, and hyperparameter optimization. Frameworks such as Auto-Sklearn, H2O.ai AutoML, and Google Cloud AutoML exemplify this approach. These systems typically employ

sophisticated search algorithms to navigate a predefined space of possible models and configurations. While highly effective, they often operate as black-box optimizers, providing limited transparency into the generated models or the rationale behind their choices. Their reliance on predefined components and search strategies can also limit their adaptability to novel or highly specialized tasks. The system presented herein differentiates itself by leveraging an LLM to dynamically generate and execute custom code, offering a more flexible and transparent approach to model construction rather than merely searching a predefined space.

The concept of LLMs acting as intelligent agents, capable of interacting with external environments and tools, has rapidly evolved. Early work demonstrated LLMs' ability to use search engines or execute simple code snippets. More recently, frameworks like LangChain and LlamaIndex have formalized the creation of LLM agents, providing abstractions for tool definition, prompt engineering, and iterative reasoning loops. These agents can perform complex, multi-step tasks by breaking them down into sub-problems, selecting appropriate tools, executing them, and integrating the results back into their reasoning process. The current system builds upon this foundation by designing a specialized set of tools that enable a Gemini AI agent to interact with a Python execution environment for the specific purpose of ML model generation. This extends the application of LLM agents to a domain requiring precise code generation, execution, and verification.

The ability of LLMs to generate executable code has been a significant breakthrough, with models like Codex and AlphaCode demonstrating proficiency in various programming languages. In the context of machine learning, LLMs have been used to assist with data analysis scripts, generate model architectures, or even write entire training loops. However, many of these applications focus on code suggestion or generation rather than autonomous execution and iterative refinement based on real-world feedback. This research goes beyond mere code generation by integrating the LLM within an active feedback loop, where it dynamically writes, executes, and debugs Python code for ML model training, adapting its strategy based on the outcomes of its actions. This closed-loop approach, where the LLM acts as a self-correcting programmer and data scientist, represents a significant step towards more autonomous ML development.

System Architecture

The proposed system is engineered as a full-stack monorepo, designed to provide a seamless user experience for autonomous ML model generation and inference. The architecture is modular, separating concerns between the frontend user interface, the backend API, and the core LLM-driven agent responsible for ML tasks.

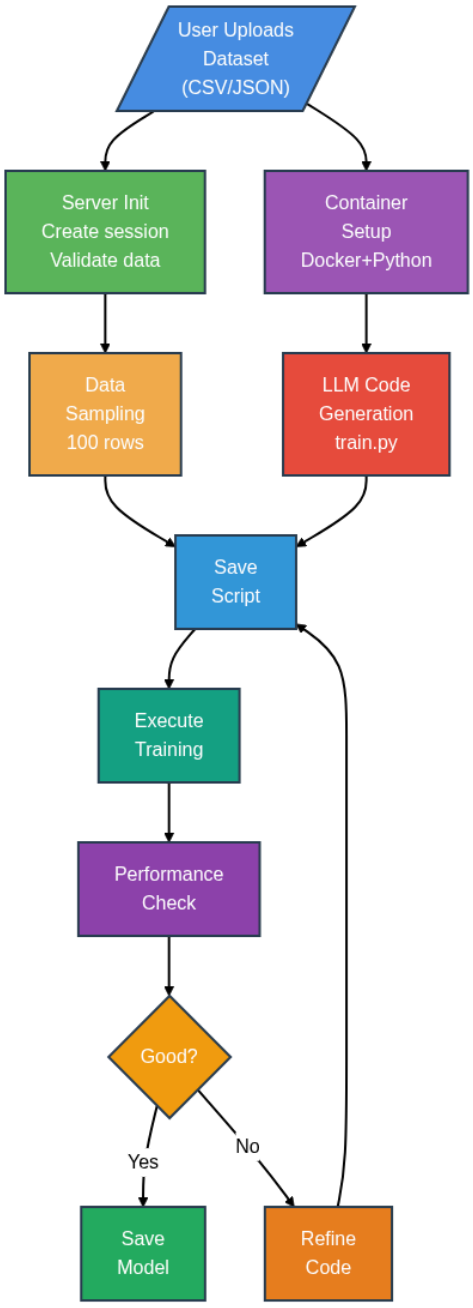
The project is organized into two primary applications: a web frontend and a server backend. The web component serves as the user-facing interface, built with Next.js and React, providing functionalities for dataset upload, initiating model training, and performing inferences. The server component, developed with Bun, Express, and tRPC, acts as the backend, handling API requests, database interactions, and

orchestrating the LLM-driven ML model generation process. This monorepo structure facilitates type-safe communication across the stack and streamlines development.

Technologies:

1. **tRPC:** Utilized for building end-to-end type-safe APIs. This ensures that data structures and function signatures defined in the backend are automatically inferred and available in the frontend, significantly reducing development errors and improving code maintainability.
2. **Prisma ORM:** Serves as the Object-Relational Mapper (ORM) for database interactions. It manages **Models**, **Inferences**, and **Datasets** collections, providing a structured way to store and retrieve information about trained models, prediction records, and uploaded data. The **ModelState** enum (**TRAINING**, **READY**, **ERROR**) tracks the lifecycle of model generation jobs.
3. **Express:** The server is built using Express.js running on Bun, providing a high-performance and efficient runtime environment for handling API requests and managing server-side logic.
4. **React:** Powers the frontend interface, enabling a responsive and modular component-based architecture. It manages user interactions, real-time updates, and dynamic rendering of model training status, inference results, and dataset management views. React hooks and context are leveraged for state management and seamless integration with tRPC endpoints, ensuring type-safe communication between the UI and backend.
5. **Langchain:** Integrates language model reasoning and orchestration capabilities into the system. It handles prompt construction, chaining, and context management for model inference workflows. LangChain enables modular composition of LLM-powered pipelines, ensuring consistent input/output formatting and easier debugging during model execution.
6. **MongoDB:** Serves as the primary database for persistent storage. It houses Models, Inferences, and Datasets collections, optimized for flexible schema evolution and fast document queries. MongoDB's indexing and aggregation pipelines support efficient retrieval of model metadata, inference results, and dataset records at scale.

Flow Diagram



Methodology and Workflow

The autonomous model generation workflow begins with a user request and is orchestrated by the **AgentService**, with the **LLMService** (powered by Gemini AI) handling the intelligent components. The process moves from dataset ingestion to the final persistence of a trained model.

User Interaction

1. **Dataset Upload:** Users upload datasets through the web interface. The backend stores them, and the **Datasets** collection (via Prisma) tracks file metadata, ownership, and storage paths.
2. **Initiate Training:** The user selects a dataset and triggers a training job. This sends a request to the backend, invoking the training route with the dataset ID.

Dataset Preparation and Context Provisioning

After receiving the training request, the **AgentService** prepares the environment:

1. **Job ID Generation:** A unique job ID is created to track the process (uuidv4).
2. **Database Record Creation:** A new entry is added to the **Models** collection with the job ID, user ID, and a state of **TRAINING**.
3. **Dataset Copying:** The dataset is copied to an isolated working directory for the job.
4. **Environment Setup:** The **UvService** initializes a Python virtual environment, ready for dependency installation and script execution.
5. **Context Provisioning:** The **LLMService** is initialized with context, specifically, the first few lines of the dataset. This gives Gemini AI insight into column names, types, and data structure for generating an appropriate training strategy.

LLM-Driven Iterative Development

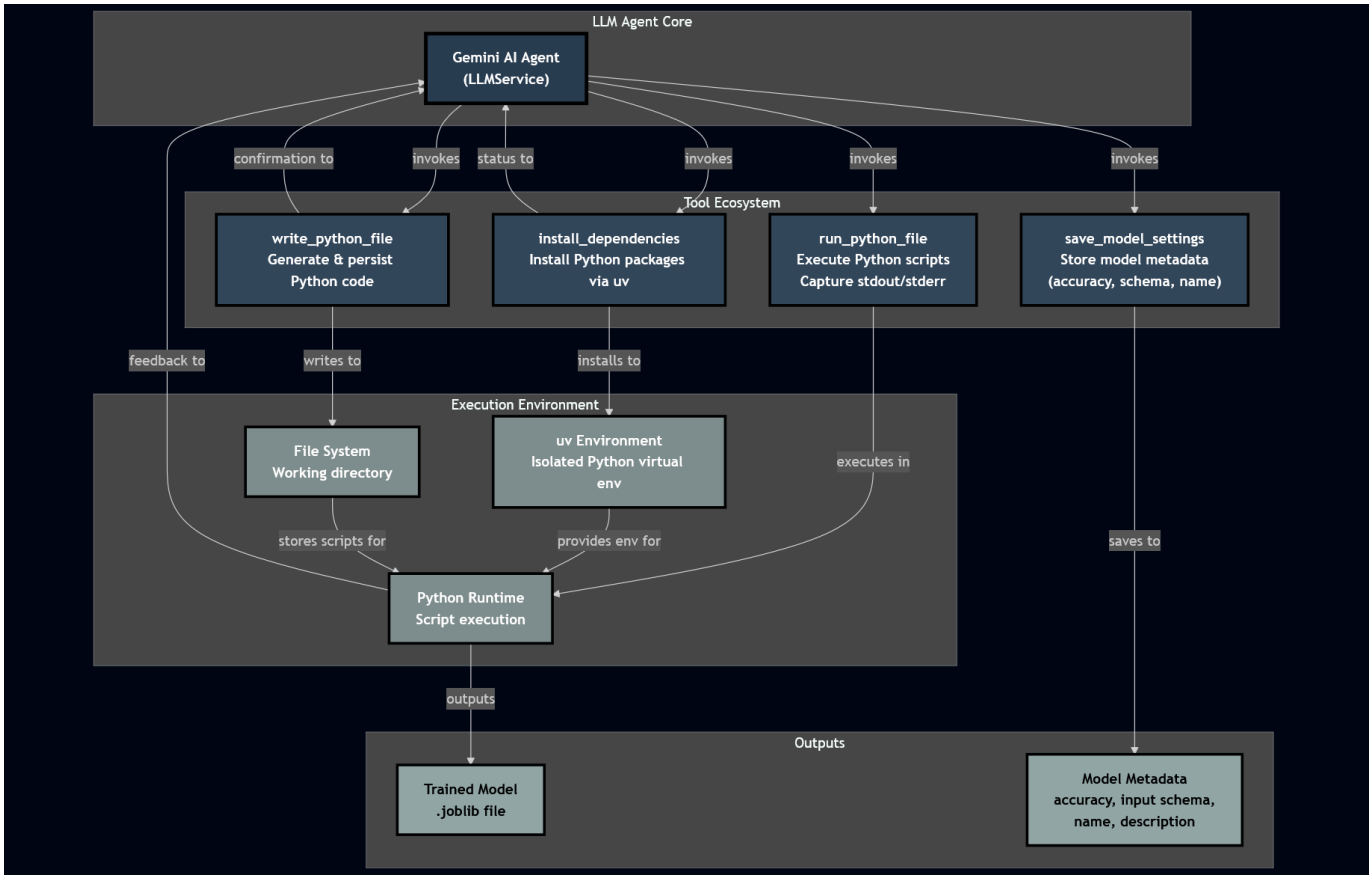
At this stage, **LLMService** manages the conversation with Gemini AI, which iteratively develops and executes an ML model.

1. **Prompt and Strategy Formulation:** Gemini AI receives an initial prompt containing task instructions and dataset context. It determines the model type, writes Python code for training and evaluation, and plans how to save the model.

2. **Tool Invocation:** The LLM may call tools like:
    - *install_dependencies* for libraries (e.g., pandas, scikit-learn)
    - *write_python_file* to generate preprocessing or training scripts

3. **Execution and Feedback:** The **LLMService** executes these tool calls and returns output (success messages, errors, metrics).

4. **Refinement Loop:** Gemini AI reviews feedback and adjusts its strategy fixing code, installing dependencies, or retraining until it achieves acceptable performance or reaches iteration limits.



(Tool Call Flow diagram)

Model Evaluation and Metadata Extraction

When Gemini AI finishes training and evaluation, it invokes the save_model_settings tool.

- The LLM provides accuracy, schema, name, and description of the model.
- **LLMService** captures this metadata to update the **Models** record accordingly.

Model Deployment and Inference

Once training is complete:

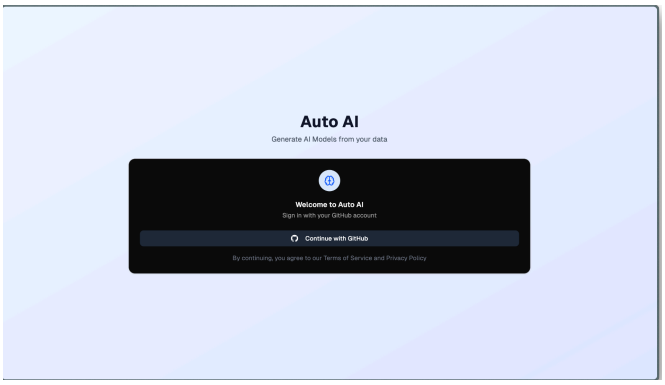
1. **Model Persistence:** The **AgentService** moves the trained model file (e.g., trained_model.joblib) to permanent storage.
2. **Database Update:** The model's status is updated from **TRAINING** to **READY**, and its metadata (accuracy, schema, description) is recorded.
3. **Inference Readiness:** The model is now available for predictions.
4. **Inference Execution:** The backend loads the trained model and performs predictions in a controlled Python environment. Results are returned to the user and logged in the **Inferences** collection.

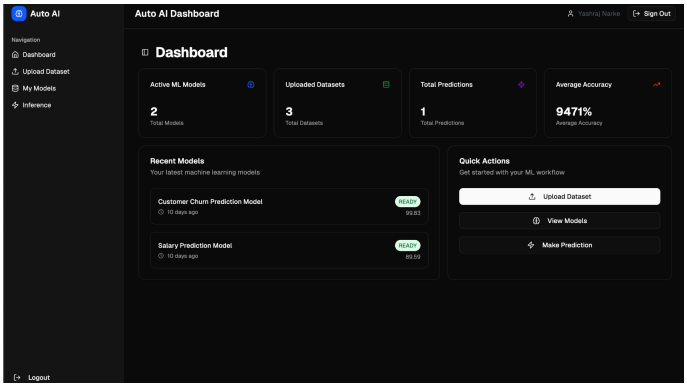
Results and Discussion

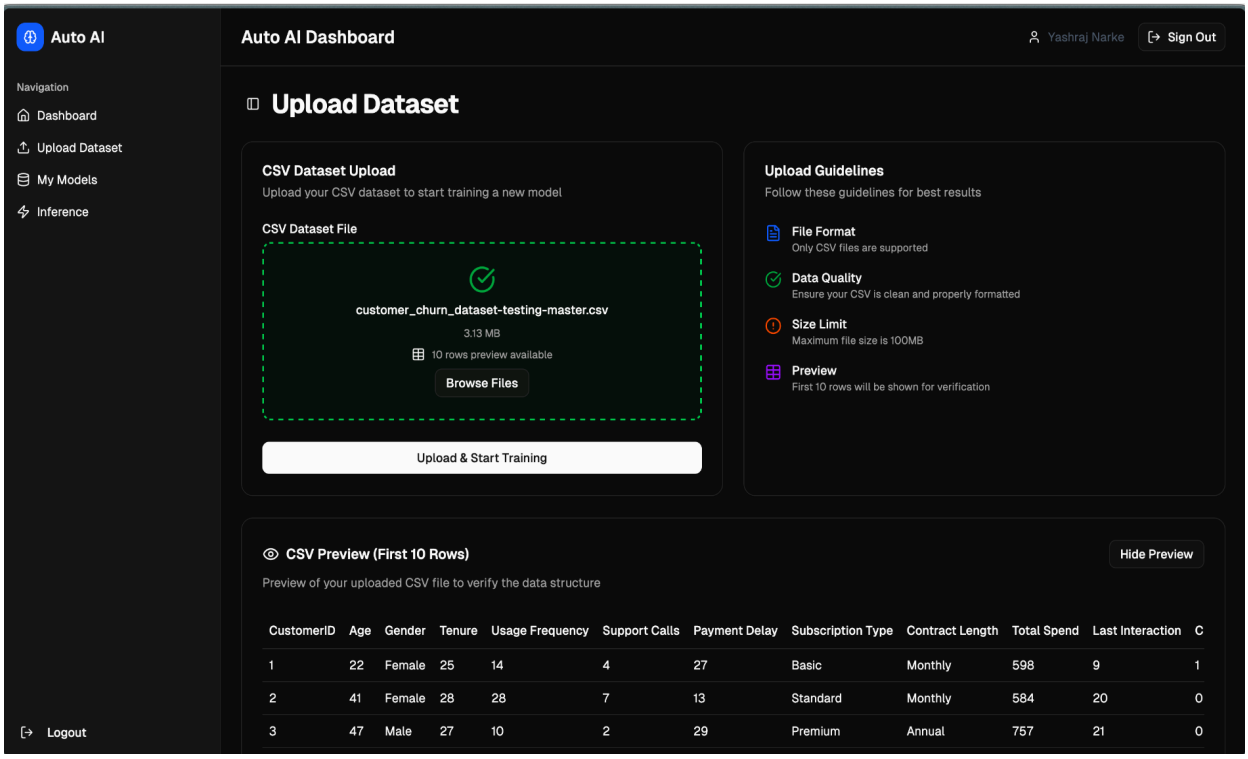
Through the orchestrated interaction between the Gemini AI agent and its tool ecosystem, the system exhibits the following key capabilities:

1. **Dynamic Code Generation and Execution**: A key capability is the LLM's ability to dynamically generate and execute Python code for data loading, preprocessing, model training, and evaluation. Given a CSV dataset, the Gemini agent can infer the need for libraries such as pandas for data manipulation, scikit-learn for model implementation (e.g., LogisticRegression, RandomForestClassifier), and joblib for serialization. It writes a Python script, executes it, and processes the output automatically.
2. **Iterative Problem Solving**: The conversation loop in LLMService enables iterative debugging and refinement. If a generated script fails due to syntax errors, missing dependencies, or incorrect API usage, the error is fed back to the LLM, which analyzes and corrects it, showcasing autonomous self-correction.
3. **Model Metadata Extraction:** After successful training and evaluation, the LLM uses the *save_model_settings* tool to extract key metadata such as model accuracy, descriptive name, and input schema. This structured output enables proper management and retrieval of models in the database.
4. **Seamless Full-Stack Integration**: The system embeds autonomous ML workflows into a user-friendly web app. Users can upload datasets and trigger training jobs without writing code. The tRPC layer ensures end-to-end type safety between frontend and backend, improving reliability and maintainability.
5. **Controlled Execution Environment**: The use of uv for managing Python environments provides isolation and reproducibility for each training job. This prevents dependency conflicts and ensures consistent, predictable execution of LLM-generated code.

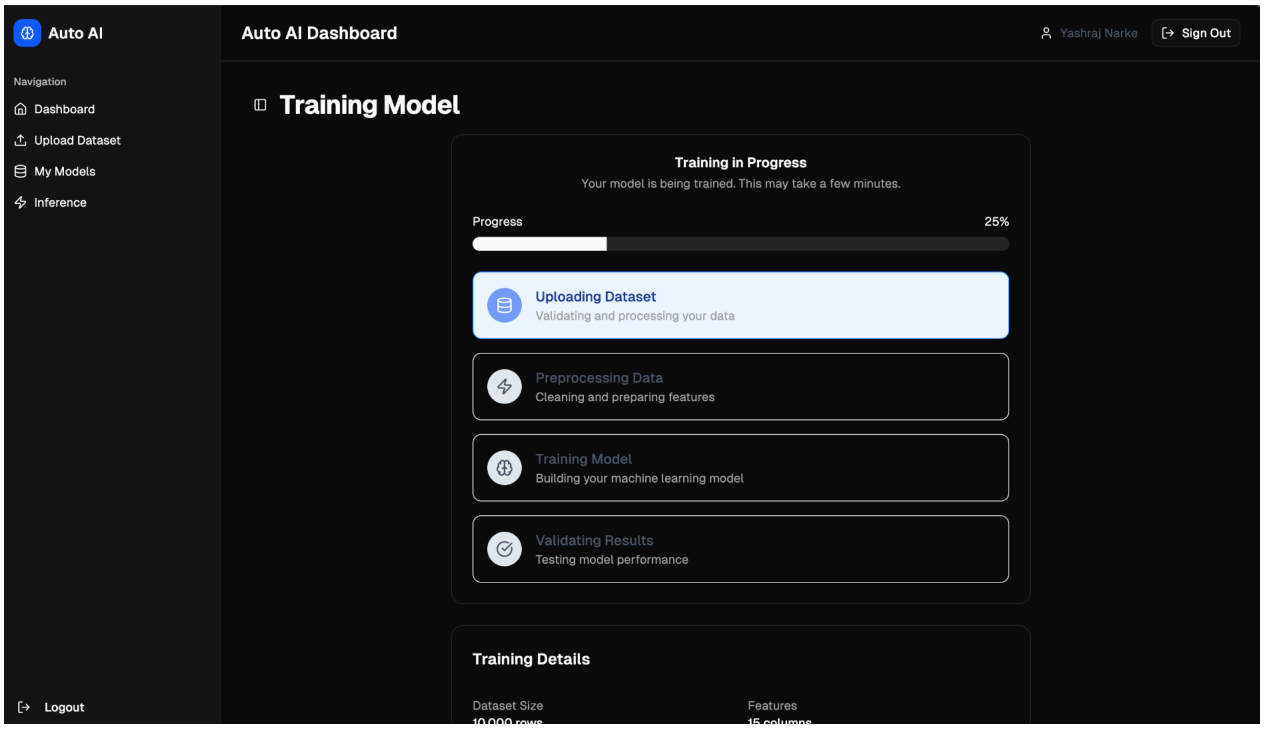
Screenshots



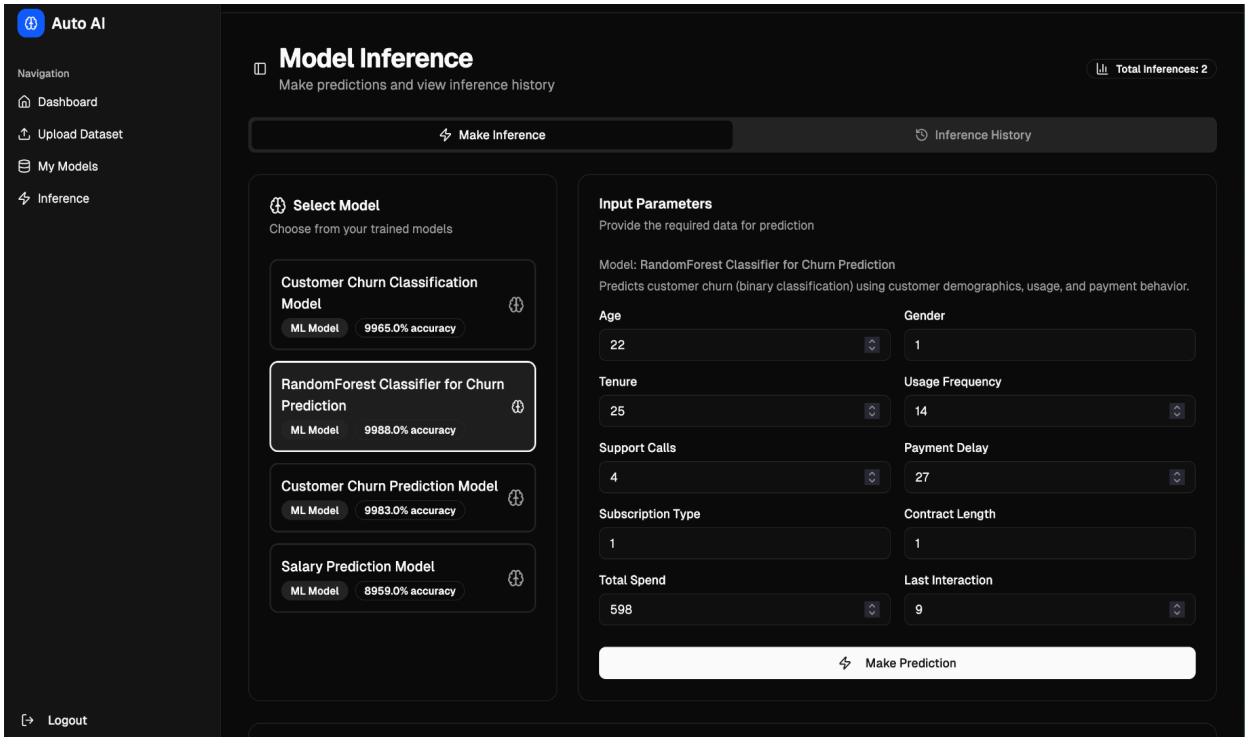




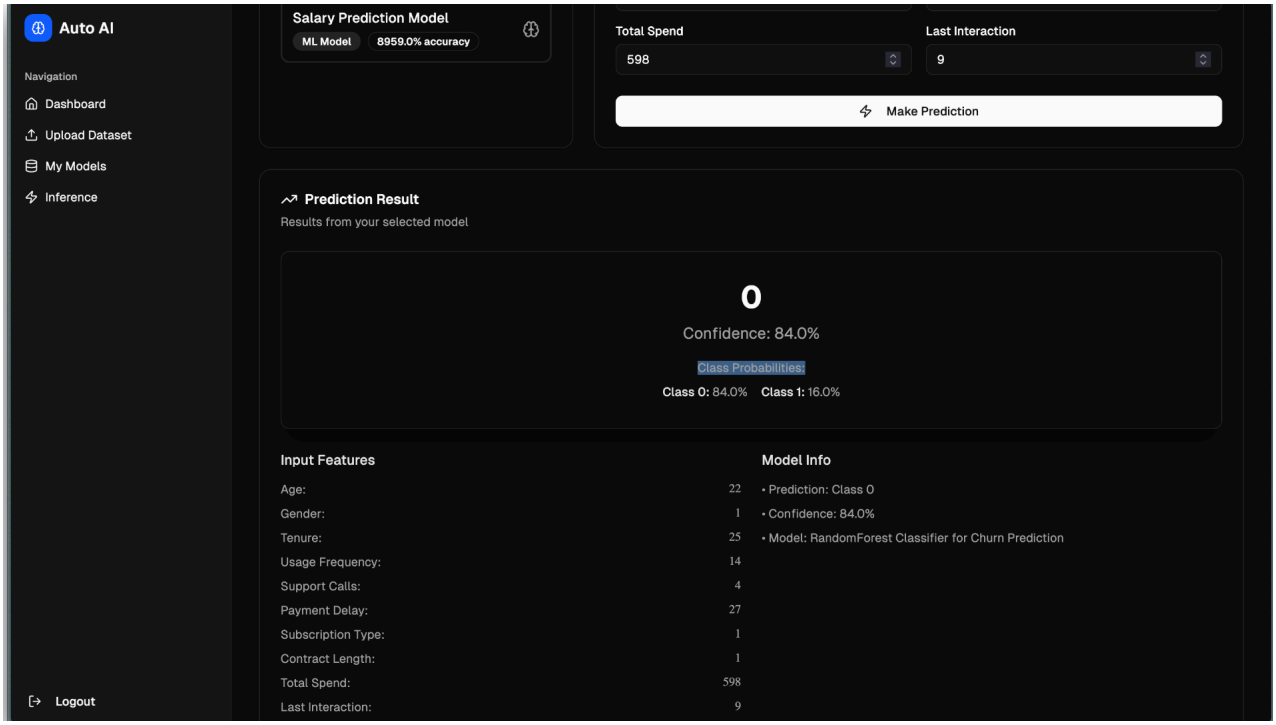
(Upload and View Dataset)



(Training Started)



(Input Inference Parameters)



(View Inference Output)

Conclusion

This project successfully demonstrates a novel approach to autonomous ML model generation by leveraging the advanced reasoning and code generation capabilities of Large Language Models (LLMs), specifically a Gemini AI-powered agent orchestrated via LangChain. The system provides enhanced flexibility and automation in ML workflows, addressing the limitations of traditional black-box AutoML solutions.

This LLM-driven AutoML paradigm promises to democratize access to advanced ML development, enabling non-experts to build efficient models with minimal manual intervention, while maintaining transparency, security, and adaptability.

References

[1] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter, "Auto-sklearn: Efficient and Robust Automated Machine Learning," in *The Springer Series on Challenges in Machine Learning*, Cham: Springer International Publishing, 2019, pp. 113–134. Accessed: Oct. 31, 2025. [Online]. Available: https://doi.org/10.1007/978-3-030-05318-5_6

[2] "H2O.ai," Convergence of the World's Best Predictive and Generative AI for Private, Protected Data. Accessed: Oct. 31, 2025. [Online]. Available: http://H2O.ai

[3] "AutoML Solutions - Train models without ML expertise," Google Cloud. Accessed: Oct. 31, 2025. [Online]. Available: https://cloud.google.com/automl

[4] A. Madaan, S. Zhou, U. Alon, Y. Yang, and G. Neubig, "Language Models of Code are Few-Shot Commonsense Learners," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Stroudsburg, PA, USA: Association for Computational Linguistics, 2022. Accessed: Oct. 31, 2025. [Online]. Available: https://doi.org/10.18653/v1/2022.emnlp-main.90

[5] "LangChain." Accessed: Oct. 31, 2025. [Online]. Available: https://www.langchain.com/

[6] "LlamaIndex," Redefine Document Workflows with AI Agents. Accessed: Oct. 31, 2025. [Online]. Available: https://www.llamaindex.ai/