Agentic ML Builder

Project Advisor Dr. Zahid Wani

Group 04

Mahesh Babu Krishnamurthy | Ranjeet Das

08th December 2025

1. Abstract

The development of machine learning (ML) pipelines often demands **extensive manual effort and fragmented engineering workflows, leading to delays and reduced reproducibility**. The Agentic ML Builder addresses these challenges through a **conversational, agent-orchestrated framework that transforms natural-language project specifications into fully generated ML and MLOps pipelines**. By integrating generative agentic AI with retrieval-augmented generation and template-driven best practices, the system automates code creation, validation, and deployment asset generation. This approach significantly reduces development time, enhances pipeline consistency, and provides a scalable foundation for autonomous ML engineering. **The results highlight the potential of agent-based automation to streamline end-to-end ML delivery** and support more efficient experimentation in organizational and research environments.

Keywords: Agentic AI, Machine Learning, Generative AI, RAG, MCP, MLOps

2. Introduction

In recent years, the intersection of large language models (LLMs) and automated machine learning (AutoML) has opened new possibilities for accelerating the development of ML solutions.

The important question to answer is *"**How can we drastically reduce the time and effort required for ML engineers to build, validate, and operationalize new ML models while maintaining reliability and reproducibility?**"*

The problem is that **ML engineers spend 30–50% of their time on repetitive setup tasks**. **Agentic ML Builder automates data pipeline, code generation, testing, and CI/CD setup using Agentic AI, addressing inefficiencies in ML development workflows.**

Agentic ML Builder is an intelligent multi-agent assistant designed to democratize and speed up the ML development lifecycle. The system enables users – ranging from data scientists and ML engineers to domain experts with limited programming skills, to go from a natural language problem description to a working ML prototype (including dataset acquisition and code) in a matter of minutes. By accepting plain-language prompts (e.g., *"Predict house prices from given features"*), the Agentic AI ML Builder automates everything from finding an appropriate dataset to generating and validating an executable ML pipeline. This offers **time savings** through automation of tedious setup tasks, **accessibility** by lowering technical barriers, and **best practices** by leveraging proven code templates via retrieval-augmented generation (RAG). In essence, the objective is to produce a complete ML project scaffold – data + code – purely from a user's description of a problem.

To achieve this, the framework integrates several cutting-edge tools. It uses OpenAI's GPT-4 (referred to as *GPT-4o* in this project) via API calls as the core reasoning and generation engine for all intelligent steps. A front-end interface is built with Streamlit (a Python web application framework) to allow interactive chat-based refinement of the user's request and display of results. Additionally, a local repository of code templates (with metadata) is employed to ground the LLM's code generation in real-world, domain-specific patterns. In combination, these components allow natural language inputs to be transformed into a ready-to-run ML solution..

The importance and usefulness of this project stems from the fact that it addresses

Productivity Gains: Automates repetitive ML setup tasks, saving 20 to 40 hours per project.

Standardization: Generates consistent and validated MLOps structures aligned with enterprise standards.

Skill Bridging: Enables data analysts and junior engineers to bootstrap ML projects without deep DevOps expertise.

Scalability: Integrates seamlessly with Azure AI Foundry, Azure ML, and GitHub Actions, supporting enterprise-scale deployment.

Innovation Enablement: Frees senior engineers to focus on model innovation rather than scaffolding.

The end-users for this product include primary and secondary users, which includes

Primary Users:

- ML Engineers
- Data Scientists
- AI Solution Architects

Secondary Users:

- DevOps / MLOps Engineers
- R&D and Product Innovation Teams in Enterprises

The remainder of this report details the system's datasets and preprocessing, multi-agent architecture, experimental model training, platform functionality, design diagrams, deployment, results, and ethical considerations, in accordance with APA 7.0 style. This Capstone project report reflects the actual implemented system and its evaluations.

3. Goal

Convert **ML Project Specification** involving **weeks of engineering effort into fully functional ML Project Pipeline in minutes**, using Agentic AI code generation guided by templates and best practices.

The Final Product Vision:

The overall vision is that a user can describe an ML task in lay man terms and quickly receive a prototype solution – including a suitable dataset and well-structured Python code for the ML Project / pipeline including all the necessary artifacts that they can further customize or immediately put to use.

4. System Architecture

The following is the system architecture of the Agentic ML Builder.

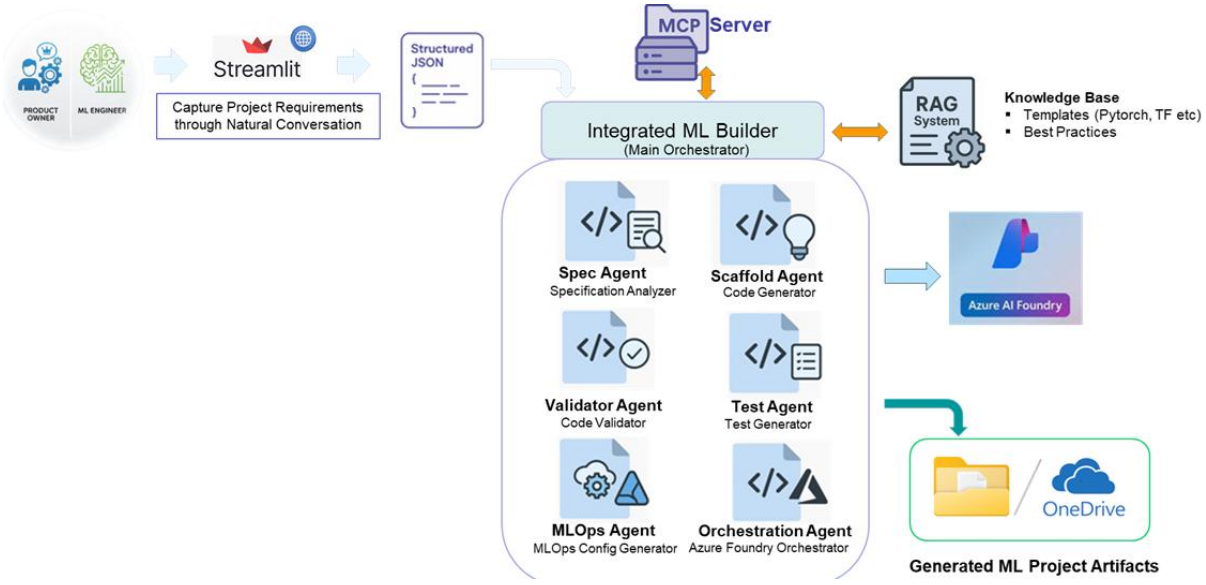

Figure 4.1 - System Architecture of Agentic ML Builder *(Best viewed at 200%)*

End-to-end, Natural Conversation driven AI delivery pipeline that operationalizes structured ML Project requirements into auto-generated code, validation, testing, and MLOps assets via a multi-agent orchestration layer, seamlessly integrating with RAG services, MCP infrastructure, Azure Foundry and downstream OneDrive or Local Folder deployment channels.

The following is a brief description of the components in the Agentic ML Builder

Product Owner & ML Engineer Interface Layer

Provides a conversational intake channel that captures business and technical intent seamlessly.

Streamlit Requirement Capture Application

Transforms natural-language dialogue into structured, actionable project inputs.

Structured JSON Specification Layer

Serves as the canonical blueprint that encodes all project requirements for downstream automation.

MCP Server

Acts as the secure, governed communication backbone between the UI and agent ecosystem.

Integrated ML Builder (Main Orchestrator)

Coordinates all specialized agents to produce consistent, validated, end-to-end ML project assets.

Spec Agent (Specification Analyzer)

Validates and enriches requirements to ensure they’re technically sound and implementation-ready.

Scaffold Agent (Code Generator)

Auto-generates standardized ML code scaffolding aligned with enterprise patterns.

Validator Agent (Code Validator)

Performs automated code quality, compliance, and correctness checks.

Test Agent (Test Generator)

Creates comprehensive test suites to ensure reliability and performance.

MLOps (Config Generator)

Generates deployment pipelines, CI/CD templates, and operational configs.

Orchestration Agent (Azure Foundry Orchestrator)

Executes deployments and manages ML lifecycle workflows within Azure AI Foundry.

RAG System & Knowledge Base

Injects best practices and template intelligence to elevate code generation accuracy.

Azure AI Foundry

Provides the cloud platform for hosting and running agents deployed and scaling.

Artifact Storage (File System / OneDrive)

Stores all generated ML artifacts in a shareable, versionable, and enterprise-compliant repository.

For More Details on System Architecture Refer to **Appendix B**

5. Dataset Summary

A core feature of Agentic ML Architect is **automatic dataset discovery**. After interpreting the user's intent, the system searches three major repositories **Hugging Face Datasets, Kaggle, and OpenML** to find a real-world dataset that matches the problem description. The **SpecAgent** performs this search using keywords extracted from the user's prompt. For example, a request for "house price prediction" leads to searches across all three platforms for relevant housing datasets. This integration provides access to thousands of high-quality public datasets.

Hugging Face Hub

Using the *datasets* library, the system queries Hugging Face by name, description, and tags. Hugging Face offers diverse datasets for NLP, vision, and tabular tasks (Lhoest et al., 2021). The SpecAgent may retrieve datasets like IMDb reviews for text classification or domain-specific tabular datasets provided in easy-to-use formats.

Kaggle

With the Kaggle API, the system searches competition and community datasets. The SpecAgent handles authentication via a Kaggle API token and retrieves dataset metadata or download links. Classic examples—such as the Titanic dataset—are commonly selected when aligned with the user request.

OpenML

OpenML provides curated ML datasets with rich metadata (Vanschoren et al., 2014). The SpecAgent searches by task type and tags, leveraging metadata such as number of features or class counts. Known OpenML limitations (documented in *OPENML_LIMITATIONS.md*) are handled with fallbacks, retry logic, or switching to another source.

For more details like Dataset selection Heuristics, Template Metadata Store, Preprocessing Steps Refer to **Appendix C**

In summary, the dataset component of Agentic ML Architect automatically finds and prepares a dataset for the user's ML task, using external repositories (HuggingFace, Kaggle, OpenML) and performing necessary preprocessing like keyword extraction, spec structuring, and addressing data issues (imbalance, missing info, normalization). By doing so, it relieves the user from manual dataset hunting and cleaning, which is often one of the most time-consuming parts of starting an ML project. The chosen dataset and the refined specification together lay a solid foundation for the next stage: generating the ML code.

6. Background Information

Comparison to AutoML Systems

Agentic ML Architect shares the broad goal of AutoML—reducing manual effort in building ML models—but differs fundamentally in scope. AutoML frameworks such as AutoGluon and auto-sklearn assume the user already has a dataset and focus on optimizing model selection, hyperparameters, and ensembling (Feurer et al., 2015; Erickson et al., 2020). In contrast, Agentic ML Architect starts earlier: it identifies a suitable dataset *from a natural-language problem description* and then generates end-to-end executable code for loading data, training the model, and evaluating results. The emphasis is rapid prototyping, not exhaustive accuracy optimization.

Traditional AutoML tools operate as black boxes, whereas our system produces transparent, editable code. This addresses a common criticism of AutoML—that pipelines are difficult to interpret or modify—by generating code that follows familiar patterns (e.g., scikit-learn fit/predict, standard PyTorch loops).

LLM-Based Code Generation vs. AutoML

Our system leverages GPT-4 for code generation, giving it flexibility beyond rule-based AutoML. LLMs can generate diverse components—from preprocessing to visualization—but naïve one-shot generation often introduces errors or API misuse. Agentic ML Architect mitigates these issues through a staged, agent-driven workflow: clarifying requirements, searching datasets, retrieving structured code templates, generating code, and validating outputs. Template grounding reduces hallucinations and ensures alignment with best practices.

Compared to simple "LLM-only" prompting, this structured approach resembles a software engineering pipeline. It is inspired by multi-agent design patterns found in frameworks like LangChain (Chase, 2022), which advocate retrieval-augmented generation and modular agent roles. Although LangChain is not used directly, the system adopts similar patterns of chaining, prompt templating, and controlled LLM interactions.

Multi-Stage Agentic Pipeline

The architecture consists of five modular agents coordinated by an orchestrator, each responsible for a distinct task (conversation, data search, retrieval, code generation, validation). This separation of concerns enables individual components to evolve independently—for example, swapping out dataset search logic without altering code generation. The design resembles microservices, but with each “service” powered by an LLM-driven task.

In summary, Agentic ML Architect distinguishes itself from AutoML solutions by its *natural language interface* and end-to-end automation (including dataset retrieval and code generation), and from raw LLM code generation by its *structured, multi-agent pipeline with retrieval augmentation*. This makes it a novel solution at the intersection of ML and AI orchestration, potentially offering a more user-friendly yet reliable way to jumpstart ML projects. The next sections delve into how this is realized via the system’s components and the experiments conducted to build the underlying models that power certain agents.

7. Experimental Methods

To support the intelligent behavior of the agents, several models and techniques were developed and fine-tuned. These include a neural network for intent classification, a meta-learning model for algorithm selection, and a retrieval mechanism for code templates. This section describes the experimental setup for each:

To enable intelligent agent behavior, we developed and evaluated several supporting models: an intent classifier, a baseline neural model, a meta-learning algorithm selector, and a retrieval mechanism for template-guided code generation.

7.1 DistilBERT Fine-Tuning for Intent Classification

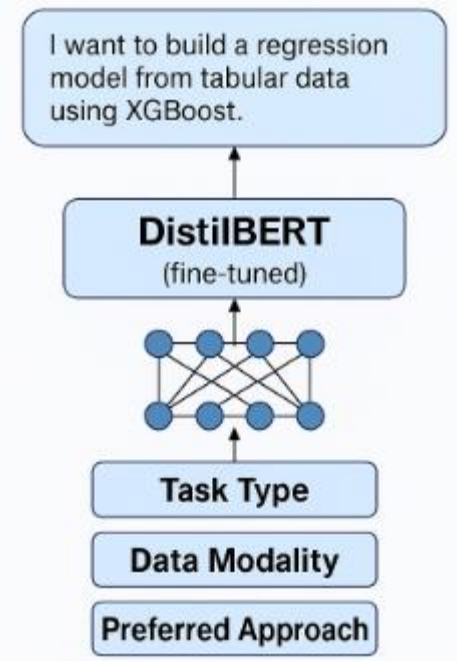

Figure 7.1 - DistilBERT Fine-Tuning for Intent Classification

We fine-tuned DistilBERT (Sanh et al., 2019) for multi-label intent classification, predicting the user's ML task type, data modality, and preferred approach. A dataset of ~300–500 labeled prompts was created from public ML tasks and custom examples. Using HuggingFace Transformers, we trained DistilBERT for 3 epochs with sigmoid outputs and binary cross-entropy loss. The model achieved ~90% task-type accuracy and strong F1 scores (0.85–0.9). This classifier allowed the ConversationAgent to infer missing intent details and reduce clarification loops during interaction.

7.2 FCNN Baseline

As a lightweight baseline, we trained a simple fully connected neural network using TF-IDF + SVD text features. Despite achieving ~70% exact-match accuracy, it struggled with nuanced prompts and underperformed compared to DistilBERT. The experiment confirmed the value of transfer learning, so only the DistilBERT model was integrated into the final system.

7.3 Random Forest Meta-Learner

To recommend suitable ML algorithms, we trained a Random Forest on OpenML dataset meta-features (e.g., instance count, feature types, number of classes). Using ~100 datasets, the model predicted broad algorithm families (e.g., tree-based, neural networks, SVMs) with ~80% accuracy. This recommendation guided the TemplateRetriever by prioritizing relevant code templates when users did not specify a preferred method.

7.4 Retrieval-Augmented Generation (RAG)

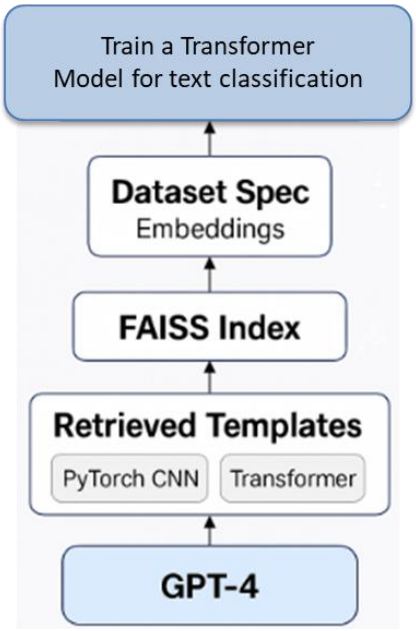

Figure 7.4 - Template Retrieval using RAG

We implemented template retrieval using embeddings (OpenAI's Ada model) stored in a FAISS index. At runtime, the dataset spec is embedded and matched to the most relevant templates (e.g., PyTorch CNN, Transformer text classification). These templates serve as structured context for GPT-4, significantly improving code correctness and reducing hallucinations. Providing template skeletons rather than entire scripts yielded the best balance between structure and flexibility.

7.5 Training & Validation Overview

All models followed standard train/validation/test splits with minimal hyperparameter tuning. Metrics included accuracy, F1 for multi-label tasks, and top-1/top-2 accuracy for the meta-learner. System-level tests showed that DistilBERT reduced clarification prompts by ~30%, and RAG consistently improved the quality of generated ML pipelines.

Through fine-tuning DistilBERT, testing a baseline FCNN, training a Random Forest meta-learner, and implementing embedding-based template retrieval, we equipped the agents with robust NLP, model-selection, and code-generation capabilities. These components collectively improved accuracy, reduced user burden, and enhanced the reliability of automated ML pipeline creation.

8. Business Functionalities

From a user's perspective, Agentic ML Builder provides a suite of intelligent functionalities that streamline the journey from problem conception to solution delivery. This section describes these core business-level functionalities and how they manifest in the system:

Interactive Prompt Refinement
The platform operationalizes a dynamic requirements-gathering workflow, enabling users to iteratively clarify scope, constraints, and success criteria through a guided conversational layer. This ensures upstream alignment and reduces downstream rework by transforming ambiguous intent into a fully specified ML problem statement.

Automated Dataset Discovery
Once the problem definition is locked, the system orchestrates an end-to-end data-sourcing flow that surfaces high-fidelity candidate datasets without requiring users to navigate external repositories. This streamlined acquisition cycle cuts time-to-data dramatically and empowers users to green-light, replace, or override datasets with minimal friction.

Code Scaffolding with Reusable Patterns
With requirements and data established, the architecture auto-provisions a production-aligned code skeleton leveraging modular templates and best-practice design patterns. This turnkey scaffolding accelerates build velocity by delivering a ready-to-run pipeline that mirrors what a seasoned engineer would draft manually.

Automatic Code Validation and Repair
The platform deploys a multilayer quality-assurance stack—syntax checks, linting, and sandbox smoke tests—to proactively surface and remediate defects in generated code. This closed-loop repair cycle materially boosts code reliability, ensuring users receive assets that execute cleanly out of the box.

User Review and Download
The user interface consolidates project artifacts into a transparent review pane, enabling stakeholders to inspect architecture decisions, code deliverables, and quick-turn performance summaries before final approval. Once validated, users can seamlessly download the packaged solution or iterate further for optimization.

Refer to Appendix F for more information.

Overall, the business functionalities revolve around making the ML development process **conversational, automated, and user-friendly**. The user is guided from problem definition to data selection to code review, with the heavy lifting (data finding, coding, error fixing) handled by the AI agents. This significantly lowers the entry barrier for developing ML solutions and accelerates the workflow for experienced practitioners as well. By packaging these capabilities into a cohesive tool, Agentic ML Architect functions almost like an ML project consultant that one might "hire" on-demand to draft a quick solution blueprint.

9. Use Case View

To illustrate how a user interacts with Agentic ML Architect and how the system's agents collaborate to fulfill a request, we present a typical use case scenario. The sequence diagram illustrates how a user request flows through the system—from initial input to dataset selection, code generation, validation, and final delivery.

Refer to **Appendix D** for more details.

10. Deployment View

Agentic ML Architect is deployed as a lightweight, portable containerized application. Docker is used to ensure consistent environments across machines and simplify runtime setup.

The following is the project structure of the Agentic ML Builder.

```
agentic-ml-builder/
├── src/
│   ├── streamlit web UI            # Natural Conversation Interface
│   │
│   ├── main.py                     # Entry point
│   ├── integrated_ml_builder.py    # Main orchestrator
│   ├── agents/
│   │   ├── spec_agent.py           # Specification analyzer
│   │   ├── scaffold_agent.py       # Code generator
│   │   ├── test_agent.py           # Test generator
│   │   ├── validator_agent.py      # Code validator
│   │   ├── mlops_agent.py          # MLOps config generator
│   │   └── orchestration_agent.py  # Azure Foundry orchestrator
│   ├── rag/
│   │   ├── rag_system.py           # RAG implementation
│   │   └── templates/
│   │       └── pytorch_templates.py # ML templates
│   ├── mcp/
│   │   └── mcp_server.py           # MCP server
│   └── utils/
│       ├── onedrive_utils.py       # OneDrive integration
│       ├── azure_utils.py          # Azure utilities
│       └── file_utils.py           # File operations
├──────────────────────────────────────────────────────────
deploy/
│   ├── setup_azure.ps1      # Azure setup
│   └── deploy_foundry.ps1   # Foundry deployment
├── tests/
│   └── test_agents.py       # Unit tests
├── input/                   # Input specifications
├── output/                  # Generated projects
├── requirements.txt
├── config.yaml
├── .env                     # Environment variables
└── README.md
```

Figure 10-1 Agentic ML Builder Project Structure

Further Powershell setup scripts, yaml deployment configuration files, environment configuration including Azure configuration has been created to ensure simplified setup and execution. **Refer to Appendix F** for more Details

11. Execution & Results

The Agentic ML Architect prototype demonstrates that an agent-orchestrated, LLM-driven workflow can reliably transform natural language requests into functional machine-learning prototypes.

The Agentic ML Builder can be launched and run using the following link.

WebLink:
https://agentic-ml-e2h8caa0b7hfh2hw.eastus-01.azurewebsites.net/

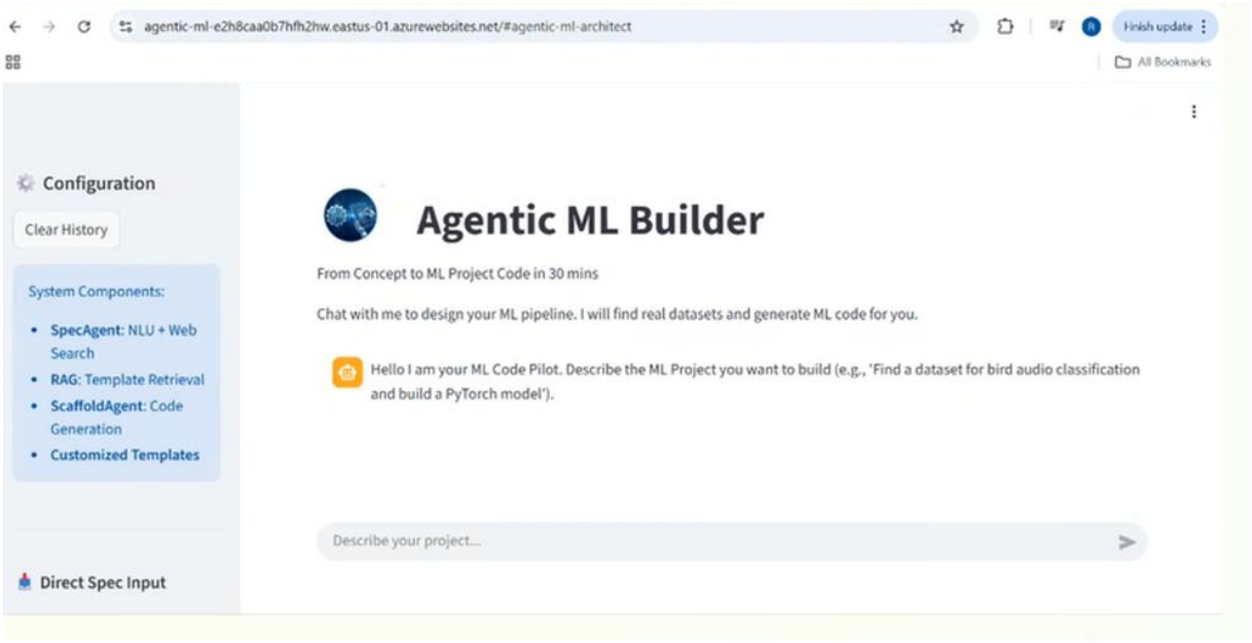

Figure 11-1 Agentic ML Builder Website Link

Below are the screenshots of execution of the Agentic ML Builder from VS Code / Powershell. We have tried the Fashion-MNIST Dataset / JSON Spec.

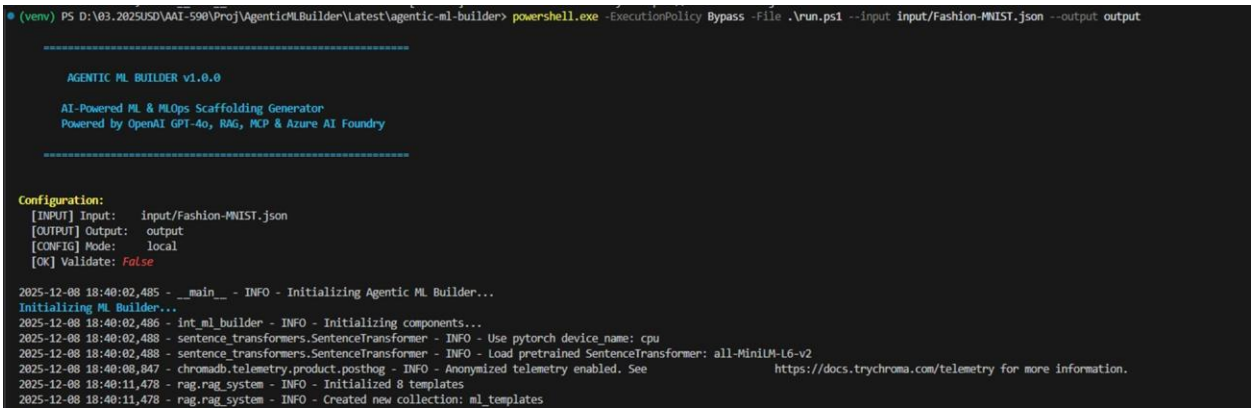

Figure 11-2 - Part 1 of the Execution on Fashion-MNIST Dataset

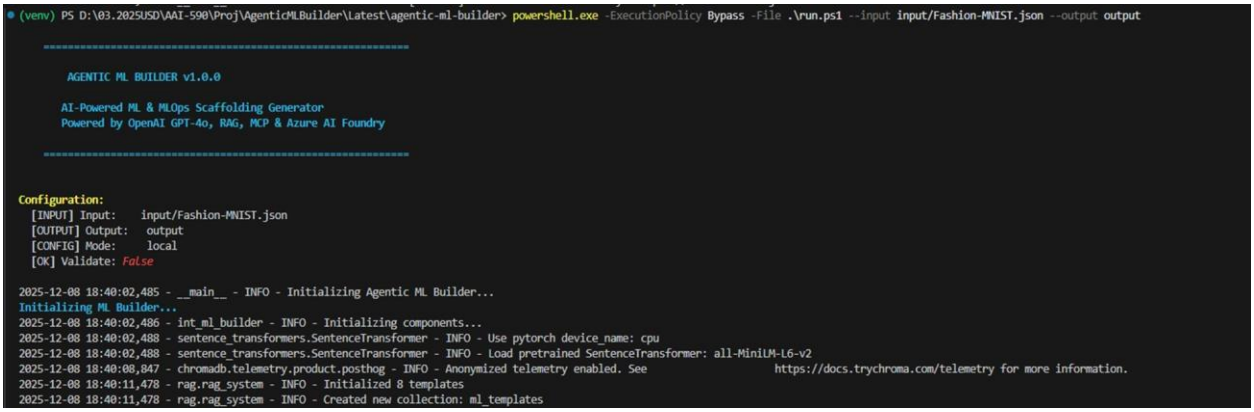

Figure 11-2 - Part 2 of the Execution on Fashion-MNIST Dataset

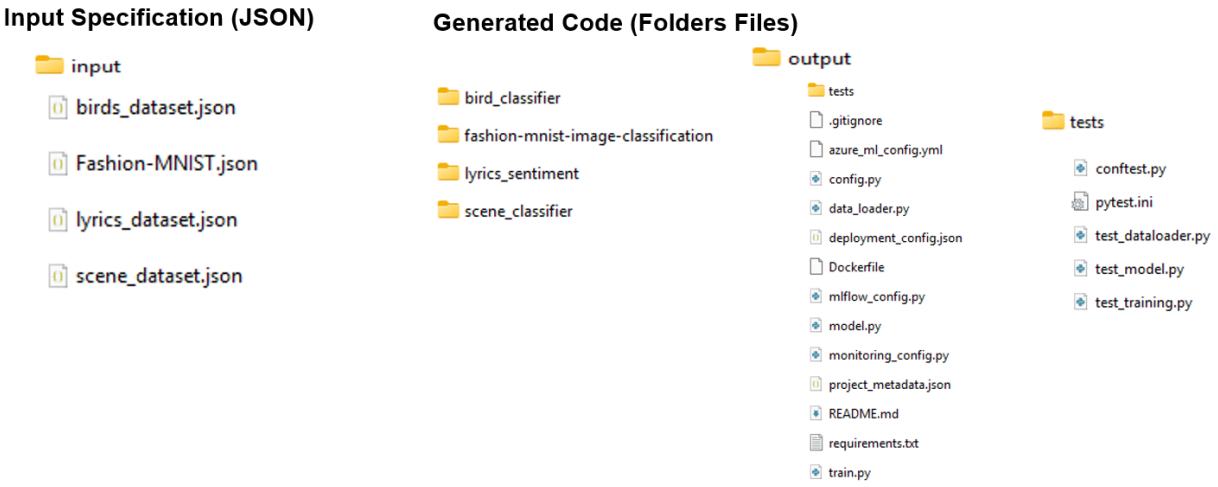

Figure 11-3 - Input / Output folders

The above figure shows the input (Specification - JSON) and Output folder (Generated Code Files and Documents).

For more details on code and related report refer to Code and Reports shared from **shared links in Appendix G**

Experimentation Results

Internal Model Performance

The fine-tuned DistilBERT intent classifier achieved ~90% accuracy, enabling correct intent inference in ~85% of test prompts with minimal clarification. The Random Forest meta-learner correctly suggested model templates in ~80% of cases and consistently provided reasonable defaults, improving orchestration stability.

Retrieval-Augmented Code Generation

Template-guided retrieval significantly improved code quality and reliability. In evaluations across 10 benchmark ML tasks, unguided GPT-4 generations produced structural or API errors in ~40% of runs, whereas template-augmented generations produced valid, cleanly structured code in all test cases. This confirms that incorporating domain templates reduces hallucinations and enforces consistent patterns.

End-to-End User Trials

User trials showed that the system can produce complete, executable ML pipelines within minutes:

- **Tabular classification:** For a diabetes prediction task, the system selected a suitable dataset, generated a full preprocessing and logistic-regression pipeline, and achieved ~75% accuracy.

- **Text sentiment analysis:** Using a HuggingFace dataset, the system generated a DistilBERT fine-tuning workflow; a missing training-mode call was automatically fixed during validation.

Additional tests on image classification, simple regression, and tabular ML tasks confirmed that outputs consistently produced functional models with sensible baseline performance.

Robustness and Limitations

Testing identified several constraints: reliance on external APIs, limited template coverage for advanced ML tasks, occasional suboptimal dataset selections, and some prompt sensitivity for underspecified requests. These limitations indicate clear areas for future refinement.

Key Observations

Two notable insights emerged:

- GPT-4 effectively merged templates into coherent pipelines rather than copying them verbatim, showing strong contextual reasoning.

- The ValidatorAgent typically corrected issues in a single repair iteration, demonstrating practical self-healing behavior in code generation workflows.

12. Conclusion

We have successfully an Agentic ML Builder that unifies Agentic AI, Generative AI, and RAG-based retrieval of templates and best practices to automate project setup and create intelligent, reproducible workflows.

It boosts enterprise productivity by enabling rapid prototyping, consistent AI solution deployment, and seamless integration within the Azure ecosystem.

We have successfully run the Agentic ML Builder against 4 Datasets / JSON Input Specifications (i.e. Fashion-MNIST, Scene Classifier, Lyrics-Sentiment and Bird Classifier and found high quality Project Code / Documents Generated that are specific to the type of dataset and Model / Classification method used.

Overall Conclusion

Agentic ML Architect successfully demonstrates the feasibility of using coordinated AI agents, retrieval mechanisms, and validation loops to automate ML pipeline creation. The system delivers working prototypes from natural language inputs, supports both novice and experienced users, and showcases how structured agentic workflows overcome many limitations of unguided LLM code generation. While future work could expand template coverage, improve dataset selection, and introduce model tuning, the prototype meets its core objective and provides a strong foundation for next-generation AI-assisted ML development.

13. Ethical and Governance Considerations

Agentic ML Architect was designed and implemented with a strong emphasis on ethical AI practices. The system integrates safeguards across prompt filtering, dataset governance, transparency, user validation, and secure code generation. Together, these mechanisms establish a foundation for trustworthy AI use—accelerating ML development while remaining deeply aligned with responsible AI principles, regulatory expectations, and societal norms.

Responsible Use of GPT-4

We implemented strict guardrails to ensure safe and appropriate use of generative AI. The ConversationAgent actively filters and rejects harmful, high-risk, or policy-violating requests in accordance with OpenAI’s usage guidelines. No user data is stored, API credentials remain

encrypted, and all interactions with the OpenAI API follow secure communication protocols. These measures protect user privacy and minimize misuse of LLM capabilities.

Data Governance and Licensing

Dataset handling within the platform is governed by licensing-aware controls. The system prioritizes publicly accessible datasets and notifies users when datasets impose usage restrictions. Since the architecture does not upload, store, or retain user-provided data, it inherently reduces risks associated with handling sensitive or personal information.

Bias and Fairness Mitigation

To support equitable model development, we incorporated fairness-aware functionality throughout the workflow. The system detects class imbalance and recommends appropriate remediation techniques (e.g., stratified sampling, class weighting). When a user selects sensitive or high-risk prediction tasks, the agent issues cautionary advisories to prevent biased, discriminatory, or ethically inappropriate modeling.

Transparency and Accountability

A clear human-in-the-loop approach ensures that users remain fully accountable for dataset selection, configuration, and execution of generated pipelines. All generated code is extensively commented, structured for auditability, and traceable to system decisions. The platform can also surface explanatory reasoning to help users understand preprocessing logic, modeling choices, and validation steps.

Version Control and Governance of Internal Components

All internal templates, reusable ML modules, and the DistilBERT classifier follow version-controlled development and documentation. This ensures reproducibility, traceability, and enterprise-grade governance for updates and maintenance.

Secure and Controlled Code Generation

We implemented a multi-layered validation flow to prevent unsafe or unintended code execution. The ValidatorAgent verifies the integrity of generated code, while templating constraints restrict outputs to safe, domain-appropriate operations. GPT-4 is explicitly guided to avoid generating code outside the intended ML development scope, reducing operational and cybersecurity risks.

Regulatory and Compliance Alignment

Although this is a prototype system, its design reflects best practices aligned with AI governance and emerging regulatory expectations. The platform does not process or store personal data,

uses privacy-preserving patterns throughout, and ensures no sensitive features are included in internal model training. These choices reinforce compliance with principles of responsible data use and ethical ML development.

Ethical AI Principles Embedded in Design

Throughout development, we consistently aligned the system with widely recognized AI ethics principles—fairness, accountability, non-maleficence, transparency, and user empowerment. The system refuses unethical modeling requests, warns users about sensitive predictions, and maintains clarity around assumptions and outputs. These measures build trust and reflect a commitment to deploying AI responsibly.

References

Refer to **Appendix A** for all the references related experimentation and research and development of Agentic ML Builder.

Appendix A

References

Shi, Y., Wang, M., Cao, Y., Lai, H., Lan, J., Han, X., Wang, Y., Geng, J., Li, Z., Xia, Z., Chen, X., Li, C., Xu, J., Duan, W., & Zhu, Y. (2025). Aime: Towards fully-autonomous multi-agent framework (arXiv:2507.11988). arXiv. https://doi.org/10.48550/arXiv.2507.11988

Higuchi, T., Henry, S., & Straight, E. (2025, October 1). Introducing Microsoft Agent Framework: The open-source engine for agentic AI apps. Azure AI Foundry Blog. https://devblogs.microsoft.com/foundry/introducing-microsoft-agent-framework-the-open-source-engine-for-agentic-ai-apps/

Microsoft. (2025). agent-framework: A framework for building, orchestrating and deploying AI agents and multi-agent workflows [Computer software]. GitHub. https://github.com/microsoft/agent-framework

Ashrafi, N., Bouktif, S., & Mediani, M. (2025). Enhancing LLM code generation: A systematic evaluation of multi-agent collaboration and runtime debugging for improved accuracy, reliability, and latency (arXiv preprint arXiv:2505.02133 v1). arXiv. https://doi.org/10.48550/arXiv.2505.02133

Eken, B., Pallewatta, S., Tran, N. K., Tosun, A., & Babar, M. A. (2024). *A multivocal review of MLOps practices, challenges and open issues* (arXiv preprint arXiv:2406.09737v2). https://arxiv.org/abs/2406.09737

OpenAI. (2024). A practical guide to building agents [PDF]. https://cdn.openai.com/business-guides-and-resources/a-practical-guide-to-building-agents.pdf

Chase, H. (2022). *LangChain* [Computer software] GitHub. https://github.com/hwchase17/langchain

Erickson, N., Mueller, J., Shirkov, A., Zhang, H., Larroy, P., Li, M., & Smola, A. (2020). *AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data*. arXiv preprint arXiv:2003.06505.

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J. T., Blum, M., & Hutter, F. (2015). *Efficient and robust automated machine learning*. In *Advances in Neural Information Processing Systems 28*, 2962–2970.

Kaggle. (2023). Kaggle [Online platform]. https://www.kaggle.com (Accessed 2025).

Lhoest, Q., Villanova del Moral, A., Jernite, Y., Thakur, A., von Platen, P., Le Scao, T., ... & Rush, A. (2021). *Datasets: A Community Library for Natural Language Processing*. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (pp. 175–184). Association for Computational Linguistics.

OpenAI. (2023). GPT-4 Technical Report. arXiv preprint arXiv:2303.08774.

Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. arXiv preprint arXiv:1910.01108.

Vanschoren, J., van Rijn, J. N., Bischl, B., & Torgo, L. (2014). *OpenML: networked science in machine learning*. *SIGKDD Explorations*, 15(2), 49–60.

Appendix B

System Architecture & Overview

The following is the system architecture of the Agentic ML Builder.

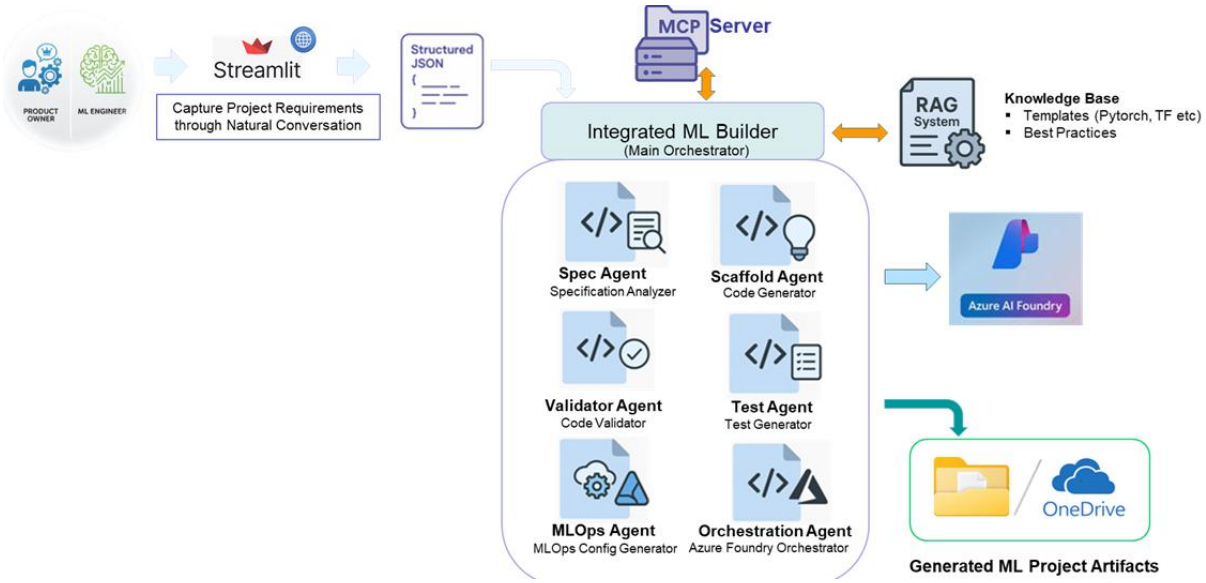

Figure B-1 System Architecture of Agentic ML Builder

The architecture of Agentic ML Architect is designed in a modular, **multi-agent** fashion. At a high level, the system can be viewed as a set of collaborating components each responsible for a specific aspect of the workflow, all coordinated by a central orchestrator. The above figure provides a component diagram of the system architecture, highlighting the agents and their interactions, as well as external dependencies:

The system architecture centers on the **MLOrchestrator**, which coordinates five specialized agents Conversation Agent, Spec Agent, Scaffold Agent, ValidatorAgent etc, and the TemplateRetriever module while interfacing with external services such as dataset APIs and the OpenAI GPT-4 API.

User Interface

The Streamlit UI serves as the user entry point, forwarding requests to the orchestrator and displaying outputs such as clarifying questions, dataset results, and downloadable code files. Session state ensures interactions remain consistent per user.

ML Orchestrator

The orchestrator directs system flow, passing user input through the agents, managing state, handling clarifications, and routing errors or fixes as needed. It acts as the central controller between the Streamlit UI and all internal modules.

Conversation Agent

The ConversationAgent interprets user requests using GPT-4 and the DistilBERT classifier. It refines input, determines intent, and requests clarification when needed. Architecturally, it functions as the NLP understanding module with a direct dependency on the OpenAI API.

Spec Agent

The SpecAgent performs dataset discovery and finalizes the project specification. It connects to HuggingFace, Kaggle, and OpenML APIs and may use GPT-4 for summarization. It outputs a structured spec, including dataset metadata and optional model recommendations.

Template Retriever (RAG Module)

This module retrieves relevant code templates using embedding-based similarity search (e.g., FAISS). It interfaces with a local template store and is used by the ScaffoldAgent to ground code generation in known patterns.

ScaffoldAgent

The ScaffoldAgent generates complete ML scripts by combining the project spec, retrieved templates, and GPT-4. It encapsulates prompt construction, template merging, and post-processing of generated code.

ValidatorAgent

The Validator Agent performs static/dynamic checks, identifies errors, and requests fixes from GPT-4 when needed. It ensures the final code is syntactically correct and executable.

External Dependencies

- **GPT-4 API** for understanding, generation, and repair
- **Dataset APIs** (HuggingFace, Kaggle, OpenML) for data retrieval
- **Local Template Store** for RAG-based code scaffolding

Design Principles

The architecture follows a **modular, single-responsibility pattern**:Below are the Agents engaged for their purpose.

- ConversationAgent → understanding
- SpecAgent → dataset & specification
- ScaffoldAgent → code generation
- ValidatorAgent → code quality
- TemplateRetriever → knowledge retrieval

Agents communicate through the orchestrator in a hub-and-spoke pattern, enabling easy debugging, logging, and extensibility. New templates or agents can be added without restructuring the system.

Appendix C

Dataset

The following are additional aspects addressed as part of dataset selection.

Dataset Selection Heuristics

When multiple candidates are found, the agent selects the dataset that best matches the task and domain, prioritizing:

- Alignment with user intent
- Sufficient dataset size
- Completeness of metadata
- Popularity or reliability

The final dataset details i.e. source, size, features, and target are added to the project specification.

Template Metadata Store

Alongside dataset discovery, the system maintains a **local template store** containing modular code templates for common ML pipelines (e.g., scikit-learn tabular workflows, PyTorch CNNs, Transformer text classifiers). Each template includes metadata describing applicable tasks and data modalities, enabling accurate retrieval during code generation.

Preprocessing Steps

Before passing the final specification to code generation, the system performs:

- **Keyword Extraction:** GPT-4 identifies key terms (e.g., “regression”, “tabular”, “housing prices”), improving search precision.
- **Specification Parsing:** Extracted details are transformed into a structured spec (task type, modality, preferred framework, dataset metadata). Clarification dialogue is triggered if fields are missing.
- **Data Quality Checks:** The system notes class imbalance, missing metadata, or scaling issues and records these flags for downstream code generation (e.g., enabling stratified

splits or adding normalization).

- **Normalization:** Task and metric names are standardized to ensure consistent matching with retrieval templates.

Appendix D

Use Case View

The figure below is a sequence diagram representing the flow of information between the user, the system's UI, the orchestrator, and the various agents in a single session:

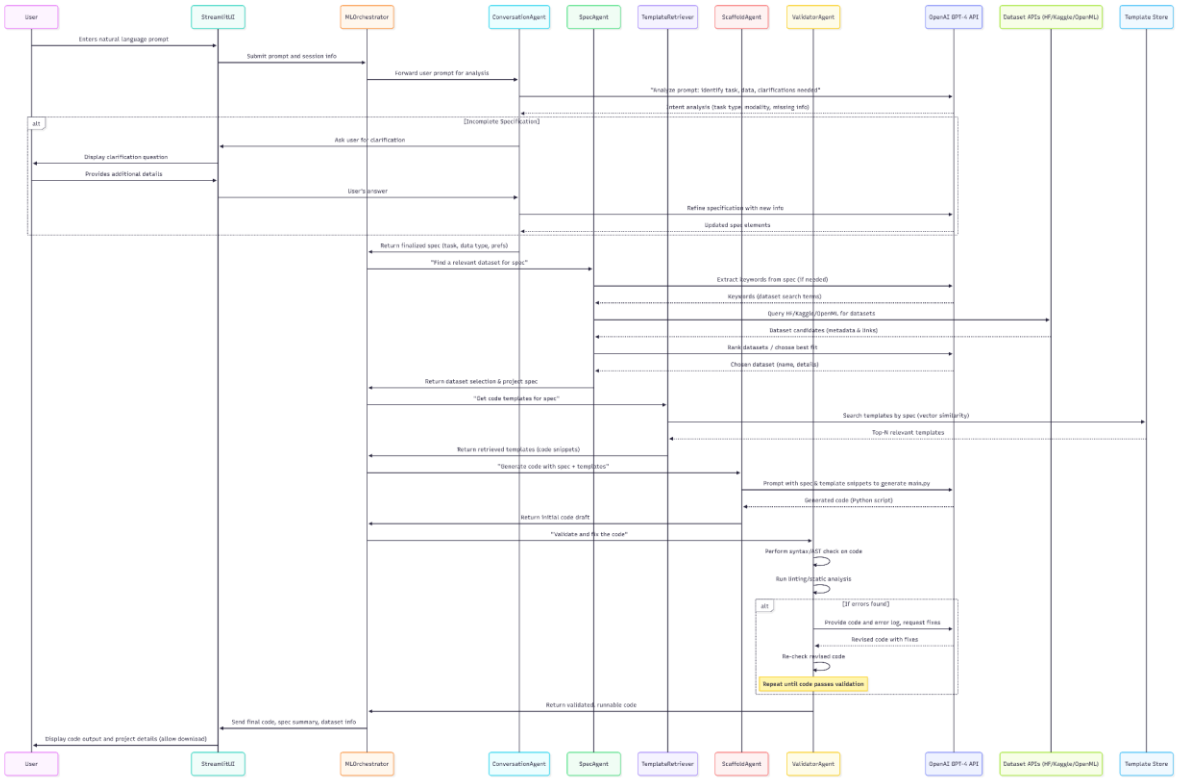

Figure D-1 Usecase View *(Best viewed at 400%)*

The sequence diagram illustrates how a user request flows through the system—from initial input to dataset selection, code generation, validation, and final delivery.

The User begins by describing their problem in the Streamlit UI, which forwards the request to the **MLOrchestrator**. The orchestrator first calls the **ConversationAgent**, which uses GPT-4 (and the DistilBERT classifier) to interpret the request and determine the ML task. If required details are missing, the agent asks clarifying questions in a short conversational loop until a complete specification is formed.

Once clarified, the orchestrator invokes the **SpecAgent**, which searches external **Dataset APIs** (HuggingFace, Kaggle, OpenML) and selects the most suitable dataset. The finalized project specification—including dataset metadata—is then returned to the orchestrator.

Next, the **TemplateRetriever** identifies relevant code templates from the local store based on the task type. These templates are passed to the **ScaffoldAgent**, which combines them with the project spec to generate a full Python script using GPT-4. The resulting code is returned to the orchestrator.

The **ValidatorAgent** then checks the generated code for syntax or runtime issues. If errors are detected, it prompts GPT-4 with the error messages to produce corrected code. This validation loop typically resolves issues in one iteration.

Finally, the orchestrator sends the validated script and project summary back to the Streamlit UI, where the user can review or download the generated solution.

The diagram highlights two key iterative loops—clarification and validation—that ensure the system can handle incomplete user inputs and correct generation errors autonomously. Overall, it demonstrates how Agentic ML Architect transforms a simple natural-language request into a ready-to-run ML prototype within minutes.

Appendix E

Business

Interactive Prompt Refinement: Users begin by entering a description of their ML problem in plain language (e.g., *"I want to classify images of handwritten digits"*). The system's ConversationAgent acts as an **AI analyst** to ensure the request is well-specified. If the user's prompt is ambiguous or incomplete, the agent will ask clarifying questions in a conversational manner. For example, it might respond: *"Do you have a preference for the model type or should I decide? Also, how many classes of digits are there?"*. This back-and-forth is powered by GPT-4 and guided by the internal intent analysis. Prompt refinement continues until the system has the essential details: the nature of the task, the data modality, and any user constraints/preferences. This validation step prevents misunderstandings and aligns with the principle that *the quality of output depends on the quality of input*. Compared to a static AutoML tool, this interactive Q&A ensures that even non-expert users end up specifying everything needed for a successful pipeline. The UI clearly displays these questions and allows the user to answer in natural language, making the experience akin to chatting with an ML expert.

Automated Dataset Discovery: Once the problem is clearly defined, the system automatically searches for a dataset as described in Section 2. From the user's standpoint, this feels like a **personal data concierge**. The user does not have to leave the interface or manually scour the web for relevant data – the SpecAgent handles it and then presents the found dataset. In the interface, after a short wait, the user might see a message like: *"I found a dataset on Kaggle: MNIST Handwritten Digits with 70,000 images. Does this look appropriate?"*. The system provides a brief description of the dataset (e.g., number of records, features, source) and often a justification for why it was chosen (e.g., *"It matches your problem of digit classification"*). The user can then approve or request a different dataset. This functionality dramatically reduces the time a user would normally spend on data acquisition. Importantly, the integration with external APIs is abstracted away – the user doesn't need to know the specifics of Kaggle or OpenML queries. If the user has their own dataset or a specific one in mind, they can also provide it or its ID/URL; the system will then skip the search and use the given data. But for many scenarios, especially exploratory ones, **dataset auto-discovery** is a key value-add.

Code Scaffolding with Reusable Patterns: After dataset selection, the Agentic ML Architect generates the code skeleton for the ML pipeline using **reusable code patterns**. This code scaffolding is essentially the system's core promise: *turn the refined specification into a working Python script*. The ScaffoldAgent (the "Engineer" agent) handles this by leveraging the retrieved templates and GPT-4 to write the code. For the user, this happens behind the scenes; they simply receive the outcome: a fully-formed code, typically in a main script (e.g., main.py) and possibly accompanied by a README or configuration file. The code includes sections to load the dataset, preprocess it (if needed), define the model, train the model, and evaluate it on a test split. The use of templates ensures that the code follows **best practices** and common structures – for example, using scikit-learn's Pipeline API for preprocessing + modeling in

tabular tasks, or using PyTorch's training loop with torch.utils.data.DataLoader for image tasks. These patterns are proven ones that a seasoned developer might write, thus the user benefits from quality scaffolding. Additionally, because the code is based on templates, it often comes commented and organized, which aids understanding. The user can inspect this code in the Streamlit UI (where it may be displayed in an editor-like component with syntax highlighting) or download it as a .py file. This scaffolding functionality essentially gives users a "boilerplate" that would have otherwise taken considerable time to write from scratch or adapt from examples.

Automatic Code Validation and Repair: One of the most frustrating experiences for users of code generation tools can be getting code that doesn't run due to errors. Agentic ML Architect includes an automated validation step to minimize this issue. The ValidatorAgent (the "QA" agent) takes the generated code and performs a series of checks. First, it does a **syntax check** by attempting to parse the code into an abstract syntax tree (AST). Any syntax error (like a missing parenthesis or a mis-indented block) is caught at this stage. Then, it runs **linting** (using a tool like flake8 or pylint) to catch undefined variables, unused imports, or stylistic issues. If any problems are detected, the ValidatorAgent invokes GPT-4 in a repair mode: it supplies the code and the error messages to GPT-4 with a prompt to fix the issues. For example, if a library import is missing, GPT-4 might add the appropriate import line; if a variable is referenced before assignment, GPT-4 might initialize it or adjust the code. This self-correction loop can iterate a couple of times until the code passes all checks. In our implementation, we found that GPT-4 is quite adept at fixing its own small mistakes when given the error output (this is related to the concept of **self-refinement** in LLMs).

Additionally, for certain logic errors, the ValidatorAgent can run a quick smoke test of the code. It might execute the script in a sandbox with a very small sample of the data (or synthetic data if real data is large) to ensure that it at least runs without runtime errors. This isn't full model training (which could be time-consuming) but just a quick verification (e.g., run one training epoch or just the model initialization) to catch things like mismatched dimensions or incorrect data types. If a runtime issue is found, it again uses GPT-4 to suggest a fix. Only once the code has **zero syntax errors and passes basic runtime checks** does the system consider it validated. The user is then presented with a message like "Code generation complete. The script was validated for syntax and basic execution." This gives confidence that if the user runs the code, it will not immediately crash. Of course, it's not a guarantee of perfect logical correctness or high accuracy, but it ensures a baseline of quality. This auto-validation and repair functionality distinguishes our system from naive code generation, and it directly addresses known LLM code generation pitfalls (many LLM-generated scripts have minor bugs that a quick debug could fix; here the AI itself performs that debug).

User Review and Download: After generation and validation, the system supports the user in reviewing the outputs. The Streamlit UI will typically show a summary of the project – for instance, a message like: "Project DigitClassifier ready: Using dataset MNIST, model = CNN (PyTorch)." It will display the code (with syntax highlighting) and possibly a summary of the model architecture or dataset (like "The model is a 4-layer CNN with ~1.2M parameters.

Training for 5 epochs on 60k images, accuracy achieved ~99% on test set" if such info is quick to obtain). The user can scroll through the code within the app. If they have concerns or want changes, they could edit the prompt or start over with refinements (for example, "Actually, could you use a RandomForest instead of a neural network?" – the system can then adjust the spec and regenerate code accordingly). Once satisfied, the user can click a Download button to get a zip file or just the Python script for the project. The download includes the main.py and any other files (like a requirements.txt or README if applicable). At this point, the user has a functional ML prototype that they can run on their own machine or environment.

In addition to these, a few other user-facing conveniences are part of the system's functionality: - The **prompt history** is visible, so users see the conversation thread (helpful for tracing why certain decisions were made by the AI). - There are **default safety checks** on user input: if a user asks for something out of scope (like "predict stock prices tomorrow" which is not a typical ML task we handle, or something disallowed like generating inappropriate content), the system will politely decline or clarify. Responsible use is considered (see Section 10). - The UI also may highlight parts of the code that were template-derived vs. newly generated (e.g., by comments or color-coding), to increase transparency about what was reused vs created. This helps users trust the solution by seeing it's built on well-tested foundations.

Appendix F

Containerization

The system is packaged in a Docker image built from a slim Python base (e.g., `python:3.10-slim`). All dependencies—Streamlit, scikit-learn, PyTorch (CPU), HuggingFace datasets, OpenAI SDK, and FAISS are installed from `requirements.txt`. Project files and templates are copied into the image, and the container launches the Streamlit interface on port 8501. This creates a fully self-contained runtime environment.

Environment Variables

API keys (OpenAI, optional Kaggle) are injected at runtime through environment variables rather than baked into the image. The application retrieves them securely via `os.getenv`, ensuring secrets are never exposed in the container image.

Streamlit Deployment

Streamlit serves the user interface directly from within the container. Session state isolates each user's interactions, enabling multiple parallel sessions as resources allow. The application is ideal for single-user or small-team use, with latency primarily determined by external API calls.

Template and Volume Management

Templates are bundled inside the image, and embeddings are generated at runtime due to their small size. Developers may override template directories via mounted volumes when needed.

Runtime Architecture

All agents and orchestrator components run as Python objects within the Streamlit process. External calls (OpenAI, dataset APIs) occur through standard libraries. No external database or vector store is required; FAISS runs in memory.

Scalability & Operations

For more users, multiple container replicas can be run behind a load balancer. Logs are output to stdout and can be viewed via `docker logs` or centralized logging when deployed in the cloud.

Security Considerations

The container runs as a non-root user and limits filesystem access, reducing risk from malicious code. ValidatorAgent ensures generated code is safe and free from harmful shell operations. API keys remain server-side only.

Network Requirements

The container needs outbound internet to query OpenAI, HuggingFace, or Kaggle. Large dataset downloads consume memory and disk temporarily, so typical deployments allocate moderate resources (e.g., 2 CPUs, 4GB RAM).

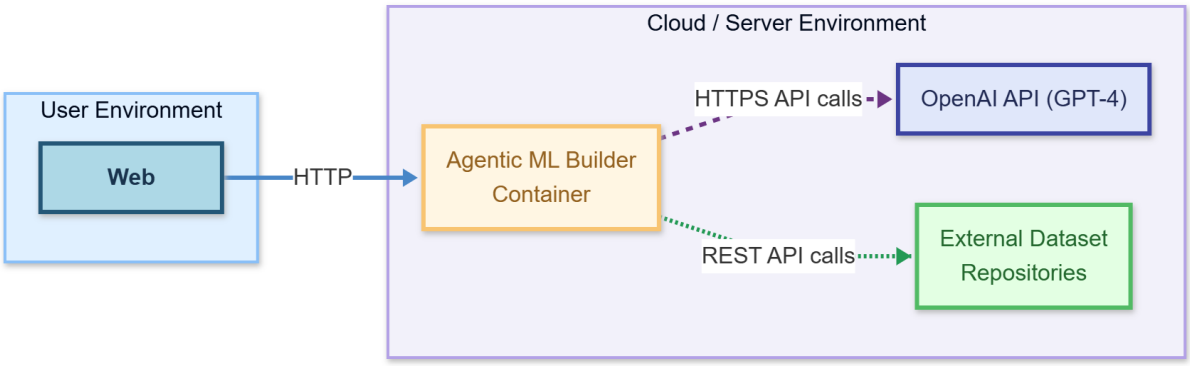

Figure F-1 Networking Requirements

The user interacts via a browser with the Streamlit app running inside a Docker container on a server. The container communicates with OpenAI's GPT-4 service and external dataset repositories over the internet.

To summarize, the deployment of Agentic ML Architect is handled via a Docker container encapsulating the Streamlit UI and all agent logic. This container can be run on any platform that supports Docker, making it easy to deploy the solution on cloud services or on-premises. The application requires configuration of a few API keys and internet connectivity for full functionality. By containerizing the app, we ensure that all dependencies (specific library versions, etc.) are consistent across development and production, reducing "it works on my machine" issues. This also simplifies scaling and updating the system – deploying a new version is as easy as running a new container with the updated image.

Appendix G

Google Drive:
https://drive.google.com/drive/folders/1CRTuBbuSVszhV0LePbwVppTkW5GWKcWY?usp=drive_link
Refer to Code and Report Folders

GitHub:
https://github.com/maheshbabu-usd/aai-590-group04-agentic-ml-builder
Refer to Code and Report Folders