Designing Data Quality Validation and Model Summarization Agents for an Agentic Clinical ML Workflow

Abstract

This document describes the design and implementation of two agents that extend an existing agentic machine learning workflow: a **Data Quality Validation Agent** and a **Model Output Summarizer Agent**. Both agents operate within a LangGraph based orchestration layer with advanced LangGraph integrations like persistence, threading, observability and live tracing for each run with LangSmith. The first agent acts as a pre-model gatekeeper that assesses data quality, detects anomalies and type inconsistencies, and produces a structured decision (PASS/WARN/FAIL) along with a detailed markdown report. The second agent consumes model performance metrics and SHAP-based feature importance, combines that with upstream data quality context, and generates a structured, human-readable summary of how the model behaves and how trustworthy its predictions are. The design is intentionally modular so that these agents can be plugged into a broader agentic workflow as described in the referenced paper, without assuming any particular downstream deployment environment.

1. Introduction

Recent work on agentic AI systems has emphasized breaking apart monolithic machine learning pipelines into cooperating "agents" that each own a well-defined part of the lifecycle: ingestion, preprocessing, modeling, evaluation, and deployment. This modular approach is especially attractive in clinical settings, where interpretability, governance, and auditability are just as important as raw predictive performance.

The case study for this project required designing two agents that are not already present in the published workflow: one focused on **data quality validation** and another focused-on **model output summarization**. Rather than treating these as isolated utilities, the goal was to integrate them into a coherent, stateful workflow where each agent's decisions influence the behavior of subsequent components.

To make the problem concrete, I chose a standard heart failure clinical records dataset with a binary outcome (death event vs survival). This dataset is clinically meaningful, tabular, and small enough to iterate quickly, while still containing the kinds of quality issues and feature relationships that make the two agents interesting to design.

2. Problem Setting and Dataset

The underlying prediction task is a binary classification problem. Each row corresponds to a patient with a set of clinical and demographic variables such as age, ejection fraction, serum creatinine, and time since follow-up, and the target column "Death Event" indicates whether the patient died during follow-up (1) or survived (0).

To properly test the behavior of the Data Quality Validation Agent, I worked with two variants of the same dataset:

- A **clean variant**, where missing values, impossible negative values, and duplicates were removed or corrected.
- A **dirty variant**, where issues were deliberately injected. In this version, certain clinical variables have high missingness (for example, serum creatinine), some non-negative quantities are made negative, the target column contains missing labels, and a subset of rows are duplicated.

Both variants share the same schema, which allows the exact same workflow to be run in two different "worlds": one where the data is "good enough" and one where it clearly is not. This contrast makes it easy to see how the Data Quality Validation Agent and the Model Output Summarizer Agent behave and how their decisions propagate through the pipeline.

3. System Overview and Architecture

The system is implemented as a small agentic workflow on top of LangGraph. All computation flows through a shared, typed state object that contains the current dataset, any diagnostic information, the model and its explanations, and the final human-readable summaries. Instead of passing large blobs of unstructured data between steps, the workflow uses **Pydantic models** to define intermediate structures such as a dataset profile and a model summary.

At a high level, the pipeline has four main stages:

1. **Data loading.** The system reads either the clean or dirty variant of the heart failure dataset, depending on a configuration flag, and stores the resulting table in the shared state.
2. **Data Quality Validation Agent.** This agent inspects the dataset, constructs a detailed dataset profile, detects issues (both rule-based and anomaly-based), and delegates to an LLM to produce a structured decision along with a markdown report.
3. **Model training and explanation.** If and only if the data quality decision is not a hard FAIL, a simple classifier (logistic regression in this implementation) is trained, evaluated on a hold-out set, and explained using SHAP to derive global feature importance.
4. **Model Output Summarizer Agent.** This agent reads the evaluation metrics, the SHAP-based feature importance, and the upstream data quality summary, and then uses an LLM with structured output to generate a narrative report.

These four stages form a linear graph in LangGraph: load data → data_quality_validation → train_and_infer → model_output_summarizer. The important point is that the edges are not just data pipes; they are **control boundaries**. For example, the training node explicitly checks the validation_status emitted by the Data Quality Validation Agent and chooses to skip modeling entirely when the dataset is deemed unsafe.

4. Data Quality Validation Agent

The Data Quality Validation Agent is the first "smart" component in the pipeline. Conceptually, it answers the question: **"Can we in good conscience train a model on this dataset, and what are the risks if we do?"**

4.1 Inputs

The agent consumes:

- The **tabular dataset** as a DataFrame-like object that has already been loaded into the pipeline state.
- The name of the target column, which in this project is inferred as "Death Event" when present.

No assumptions are made about the specific modeling algorithm downstream. The agent is designed to operate purely at the data level.

4.2 Outputs

The agent produces three main types of outputs that are written back into the pipeline state:

1. A **dataset profile**, which is a structured summary of the dataset. It includes global properties such as the number of rows, columns, and duplicate rows, and a list of per-column profiles. Each column profile stores the inferred type (numeric, categorical, or mixed), missing counts and percentages, the number of unique values, and optional numeric statistics such as mean, standard deviation, range, quartiles, and estimates of outlier counts. For the target column, the profile may also contain a simple class distribution.
2. A **data quality decision**, represented as an object containing a status field and associated explanation. The status is one of PASS, WARN, or FAIL, and is assigned by the LLM after reviewing the dataset profile and the list of issues. The object also contains a free-text, but well-structured, markdown summary and a list of specific issues.
3. A **markdown report**, which is written to disk and can be inspected independently of the pipeline. The report starts with a "Data Quality Report" heading and includes sections for an overview, column-level checks, row-level anomalies, recommendations, and an explicit list of issues.

These outputs are not just for human consumption. The status field is used programmatically by the next node in the graph to decide whether it should proceed with model training or halt.

4.3 Internal Logic

Internally, the agent is a hybrid of deterministic rules, anomaly detection, and LLM-based reasoning.

First, it runs a set of **rule-based checks**. For each column, it computes the fraction of missing values and compares this to configurable thresholds (for example, warn when more than 10 percent of values are missing, and fail the column when more than 30 percent are missing). It infers whether the column is numeric, categorical, or mixed by inspecting the data type and attempting to coerce values to numbers. Mixed-type columns are flagged as potential problems, since they often indicate inconsistent data entry.

For columns that appear numeric, the agent computes basic descriptive statistics including mean, standard deviation, minimum, maximum, and quartiles. It then uses the interquartile range to define an outlier band and counts how many values fall outside it. Columns with a large proportion of outliers are flagged; the intention here is not to assert that the values are incorrect, but to highlight that the distribution is unusually heavy-tailed or contains extreme values that may stress the model.

The agent also encodes simple domain expectations. Certain clinical variables, such as ejection fraction or serum sodium, should not be negative. For those columns, any negative values are treated as “impossible” and automatically added to the issue list. At the dataset level, it counts duplicate rows and inspects the target column for missing labels, presence of only a single class, or severe imbalance.

Second, the agent applies **row-level anomaly detection** using IsolationForest on the numeric features (excluding the target). This is meant to capture multi-dimensional oddities that the simple per-column rules might miss. After scoring all rows, the agent computes the fraction of points marked as anomalous and, if this fraction is high, appends a corresponding issue. In the dirty dataset, for example, roughly ten percent of rows are flagged as anomalous, which is a non-trivial signal that something is off. We apply Isolation Forest because traditional rule-based checks (IQR, missingness) only detect simple one-column issues. Isolation Forest is an unsupervised multivariate anomaly detector that can flag rows whose combinations of values are statistically unusual. We use it only for reporting not to remove data to give a deeper, more intelligent view of data quality. It is needed because it completes the DQ agent by adding multivariate anomaly insights that simple rules cannot catch.

Finally, all of this structured information is handed to an LLM. Instead of asking for a free-form paragraph, the prompt requests a **DataQualityDecision** object with a specific shape: it must include the status, a markdown-formatted summary report with named sections, and a list of issues. The LLM “reads” the dataset profile and the rule-based issues and then synthesizes a high-level narrative and an overall verdict. This combination allows the system to be both transparent (in terms of the numbers and flags it computes) and expressive (in terms of the final explanation that a human would read).

4.4 Rationale

Designing the agent in this way reflects several priorities. First, I wanted the agent to be model-agnostic, focusing strictly on the data. Second, I wanted it to be **explainable**: someone should be able to see not just that the dataset failed validation, but also *why*. Third, the agent needed to emit a single, simple control signal (PASS, WARN, or FAIL) that other agents could use to adjust their behavior. Combining classic data profiling, anomaly detection, and LLM reasoning provides a good balance between structure and flexibility.

5. Model Output Summarizer Agent

The second agent is placed on the other side of the modeling step. Its job is to answer a different question: **"Given this trained model, how well does it perform, which features matter most, and how confident should we be given the underlying data quality?"**

5.1 Inputs

The Model Output Summarizer Agent expects the following information in the shared state:

- The set of **evaluation metrics** produced by the model training node, including AUC, accuracy, and F1 score on a held-out test set.
- A ranked list of **global feature importances** derived from SHAP. In this project, logistic regression is used together with a SHAP explainer, and the mean absolute SHAP value per feature is used to define importance.
- The **data quality decision** from the previous agent, specifically the validation_status and the markdown summary of data quality issues.

If model training was skipped because data validation failed, the metrics and feature importance structures are absent or empty. In that scenario, the summarizer agent produces a short placeholder report that explains that no model is available.

5.2 Outputs

The main output of this agent is a **model summary** object, which is then rendered as a markdown report and saved to disk. The summary is structured into several sections: a title, an executive summary, a performance section, a feature importance section, a data quality considerations section, and recommendations.

In the executive summary, the agent is encouraged to explain, in a few sentences, what the model predicts, how well it does so, and whether there are any immediate data quality concerns. The performance section is more explicit, restating the metrics and interpreting their meaning in plain language. The feature importance section describes which features SHAP identified as most influential, and how changes in those features tend to affect the predicted risk. The data quality section explicitly references the upstream Data Quality Validation Agent's findings, and the

recommendations section translates all of this into practical guidance for how a clinician or data scientist should think about using the model.

5.3 Internal Logic

Similar to the first agent, the Model Output Summarizer Agent wraps a relatively small amount of deterministic logic around an LLM. It first assembles a compact JSON-style payload containing the metrics, the top features with their importance scores, the validation status, the raw data quality summary, and a short description of the prediction task. This payload is passed into an LLM using a structured output interface that targets a **ModelSummary** schema.

The prompt for this agent is more prescriptive about structure than about content: it describes what each section of the summary should contain and how the LLM should weave together model performance and data quality context. In particular, the LLM is asked to explicitly state that the feature importance is based on SHAP values and to mention specific feature names in its explanation. It is also asked to discuss how the data quality rating (PASS, WARN, or FAIL) might influence the confidence we have in the model's predictions.

Once the LLM returns a ModelSummary object, the agent simply formats it as markdown and writes it to both the pipeline state and a file. This makes the summary easy to display in a UI or attach to an email, while still keeping the structured representation available for further automation.

5.4 Rationale

The rationale behind this agent is to separate the responsibilities of *training* a model and *explaining* it. In many real-world systems, these two tasks are tightly coupled inside a single notebook or script, which makes it difficult to reuse explanations or adapt them to different audiences. By turning model summarization into an independent agent, the system can reuse the same summarization logic across different models, plug in alternative explanation backends (such as LIME or attention-based visualizations), or even tailor the language and level of detail to the reader.

Moreover, tying the summarizer directly to the upstream data quality decision reflects how people actually reason about models: they rarely look at performance metrics in isolation. Instead, they implicitly ask "How trustworthy are these numbers, given what I know about the data?" The agent formalizes that intuition by including data quality context in the summary.

6. Integration with the Broader Agentic Workflow

Although this project implements a compact pipeline, it is designed to fit naturally into a richer agentic architecture of the sort described in the reference paper. In a full-scale system, one would likely have separate agents for data ingestion, schema validation, anonymization, feature engineering, hyperparameter tuning, and deployment. The two agents described here would sit in that ecosystem as follows.

The **Data Quality Validation Agent** would be invoked immediately after ingestion and basic preprocessing and before any heavy modeling work. It would consume outputs from earlier agents (for example, those responsible for normalization and encoding) and then emit a decision that could either:

- Allow the pipeline to proceed, perhaps with a descriptive warning;
- Trigger a remediation subflow, where additional cleaning or imputation is performed; or
- Halt the pipeline and raise an alert to the human operator.

The **Model Output Summarizer Agent** would be invoked after the modeling and explainability agents have done their work. It does not depend on any particular model type; it only expects a dictionary of metrics and a list of feature importances. In a richer system, the inputs to this agent could easily be extended to include per-patient explanations, calibration results, or fairness metrics. Its outputs could be routed to logging, dashboards, documentation, or even external reporting systems.

By expressing both agents in terms of a shared state and well-defined schemas, the implementation remains relatively decoupled from the rest of the pipeline. They can be slotted into a larger graph without rewriting their core logic, which is exactly what an agentic design is meant to encourage.

7. Design Choices and Limitations

A few design choices are worth calling out explicitly. First, I deliberately restricted the project to **tabular data** and a single, well-understood dataset. This allowed me to focus on the agent design, the data quality logic, and the integration with LangGraph, rather than spending time handling images or text. The abstractions used, however, are general enough that additional data modalities could be added later with new agents.

Second, I chose a simple model (logistic regression) and a single explanation technique (SHAP) to keep the modeling step clear and interpretable. In principle, the training node could be swapped for a more complex model, and the summarizer agent would continue to work as long as the format of the metrics and feature importance remained consistent.

Third, both agents rely on **structured LLM outputs**. This choice reduces brittleness compared to free-form prompts, but it also assumes that the LLM is capable of respecting the requested schema. In practice, the use of a schema-aware interface makes this assumption reasonable.

There are also clear limitations. The Data Quality Validation Agent does not perform any automatic cleaning beyond detection and reporting; it is advisory, not self-healing. The workflow does not yet include an audit trail agent or an error recovery agent, both of which would be natural next steps to make the system more robust in production. The anomaly detection is also fairly simple and only works on numeric columns; more sophisticated or domain-specific detectors could be added later.

8. Conclusion

This design document has described the motivation, architecture, and detailed behavior of two agents that extend an agentic machine learning workflow in a clinically relevant setting. The Data Quality Validation Agent provides a structured, interpretable gateway that prevents modeling on obviously flawed data, while the Model Output Summarizer Agent converts raw performance metrics and SHAP values into coherent narratives that clinicians and data scientists can act on.

Together, these agents illustrate how agentic principles modularity, explicit contracts between components, and the use of language models for high-level reasoning can be applied to concrete problems in clinical ML. They also provide natural hooks for future agents, such as those for audit logging, error recovery, fairness analysis, and multimodal integration, making them a useful foundation for a richer agentic ecosystem.