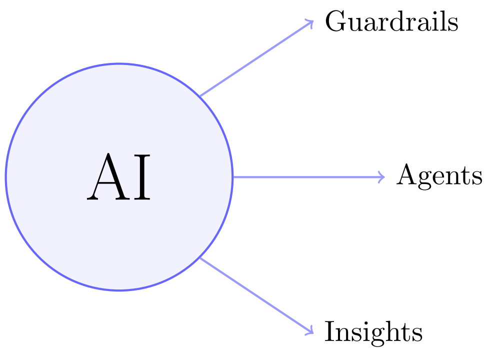
Agentic AI Framework for Predictive Analytics

A Comprehensive Analysis of Bank Customer Churn Prediction

Assignment A2 Report



December 6, 2025

Abstract

This report provides a comprehensive analysis of an agentic AI framework developed to predict customer churn using the Bank Customer Churn dataset. The system implements a multi-agent architecture featuring specialized agents for data profiling, model building, and insight generation, all orchestrated within a robust guardrail pipeline designed to ensure safety and mitigate hallucinations. We present a detailed examination of each system component, discuss the theoretical foundations of the guardrail scoring mechanisms, analyze the challenges encountered when working with free-tier API constraints, and provide a complete run-through demonstrating the system's capabilities using actual output data from Run 2.

Contents

1 Introduction

1.1 Project Objective

The primary objective of this project was to build a predictive modeling framework using **Agentic AI** principles as outlined in the Tredence article on "The Next Evolution of Predictive Analytics with Agentic AI." Rather than simply training a machine learning model, the goal was to create an autonomous system capable of:

- Understanding data structures without explicit programming
- Selecting appropriate algorithms based on data characteristics
- Answering natural language queries about risk and retention factors
- Operating safely within defined guardrails to prevent harmful outputs

1.2 Dataset Selection

We selected the **Bank Customer Churn** dataset from Kaggle, which contains 10,000 customer records with 18 features including demographics, account information, and behavioral indicators. The target variable is `Exited`, indicating whether a customer churned (1) or stayed (0). The dataset exhibits a 20.38% churn rate, presenting a moderate class imbalance challenge.

1.3 Report Structure

This report is organized as follows: Section 2 details the complete system architecture; Section 3 provides an in-depth explanation of the guardrails implementation; Section 4 discusses our experiences building the system; Section 5 covers challenges faced; and Section 6 presents a complete run-through using Run 2 outputs.

2 System Design and Architecture

2.1 High-Level Architecture Overview

The system is built as a **sequential multi-agent pipeline** that mimics the workflow of a professional data science team. The architecture consists of three specialized agents coordinated by a central orchestrator, all wrapped within a multi-layered guardrail system.

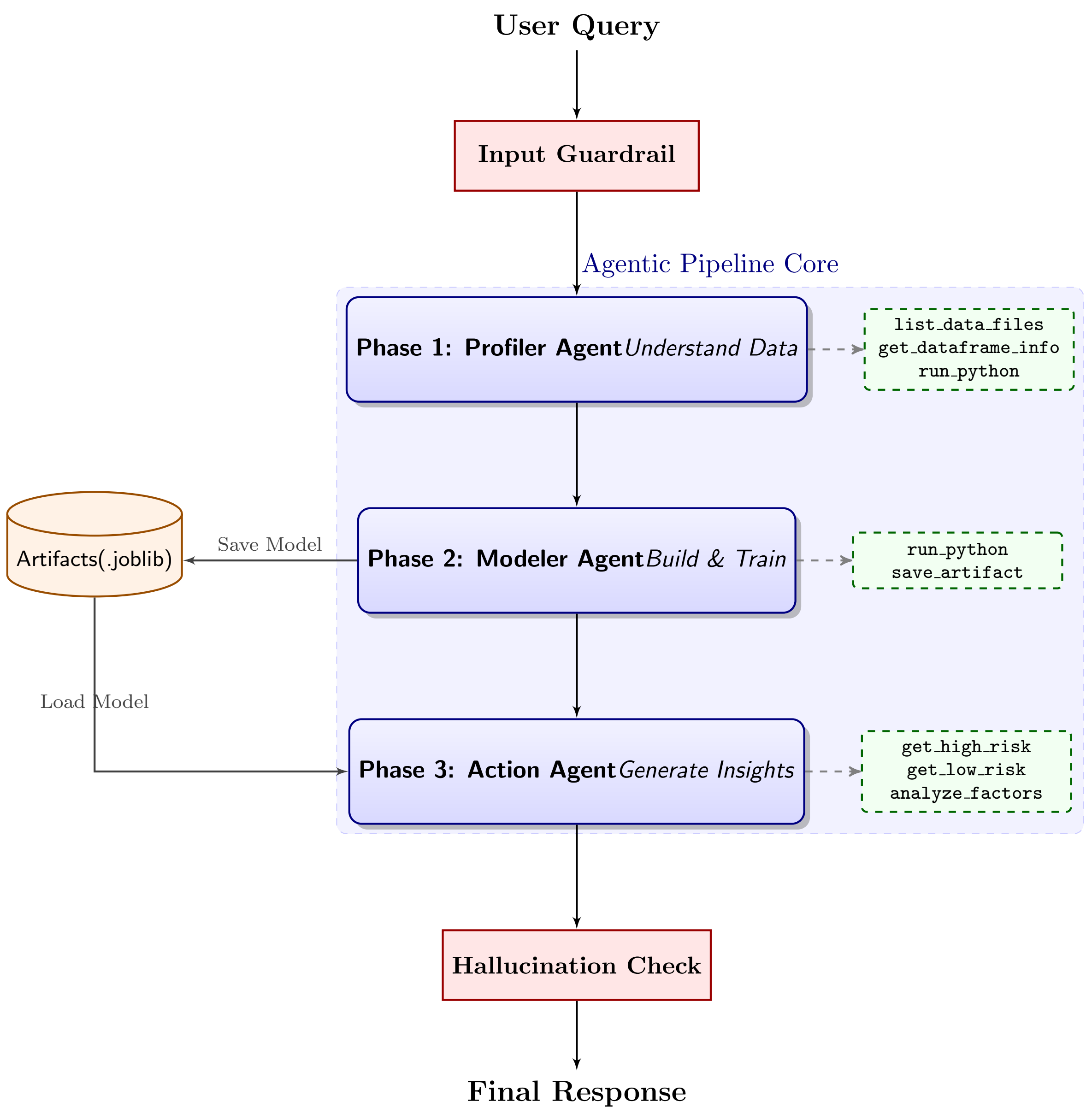


Figure 1: Agentic AI Pipeline Architecture: Orchetstrating specialized agents with tool verification and fallback mechanisms.

2.2 Agent Descriptions

2.2.1 Phase 1: Profiler Agent

The Profiler Agent is responsible for **exploratory data analysis (EDA)** and understanding the dataset structure. It operates autonomously to:

- Discover available data files in the data directory
- Analyze schema, data types, and missing values
- Identify the target variable (`Exited`)
- Calculate class distribution and churn rate
- Flag potential data quality issues

System Prompt Design: The profiler's system prompt explicitly instructs it to *not* make predictions, ensuring separation of concerns. This prevents the agent from attempting to answer user questions prematurely before a model exists.

Available Tools:

- `list_data_files`: Discovers CSV files in the data directory
- `get_dataframe_info`: Returns schema, dtypes, and basic statistics
- `run_python`: Executes custom Python code for deeper analysis

2.2.2 Phase 2: Modeler Agent

The Modeler Agent functions as an **autonomous ML engineer**. Given the profiler's analysis, it:

- Prepares features (drops ID columns, encodes categoricals)
- Trains multiple models: Logistic Regression, Random Forest, Gradient Boosting
- Evaluates each model using accuracy, precision, recall, and F1 score
- Selects the best model based on F1 score
- Persists the model to the artifacts directory using `joblib`

Fallback Mechanism: A critical design decision was implementing a **direct training fallback**. If the LLM fails to generate valid code (due to API errors or malformed tool calls), the system falls back to a deterministic Python function that performs the same training steps. This ensures the pipeline always completes.

2.2.3 Phase 3: Action Agent (Insights Generator)

The Action Agent is the **customer-facing interface**. It receives the user's original query and uses the trained model to generate actionable insights:

- Loads the persisted model from artifacts
- Makes predictions on the full dataset
- Identifies high-risk and low-risk customers
- Analyzes feature importance to explain predictions
- Generates and saves comprehensive reports

Specialized Prediction Tools:

- `get_high_risk_customers(top_n)`: Returns the N customers with highest churn probability
- `get_low_risk_customers(top_n)`: Returns the N customers with lowest churn probability
- `analyze_churn_factors()`: Extracts and ranks feature importance

2.3 Orchestration: The RobustAgent Class

A key innovation in this implementation is the `RobustAgent` class, designed to handle the instability of free-tier API services. Key features include:

Listing 1: RobustAgent Retry Logic

```python
class RobustAgent:
    def __init__(self, llm, tools, system_prompt, name):
        self.max_iterations = 8    # Cap tool calls
        self.max_retries = 3       # Retry on API failure

    def invoke(self, task):
        for iteration in range(self.max_iterations):
            time.sleep(RATE_LIMIT_CONFIG["retry_delay"])
            for retry in range(self.max_retries):
                try:
                    response = llm_with_tools.invoke(messages)
                    break
                except Exception as e:
                    # Exponential backoff
                    time.sleep(2 ** retry)
```

Rate Limiting Configuration:

- `requests_per_minute`: 25 (under Groq's 30 RPM limit)
- `tokens_per_minute`: 5000 (under Groq's 6000 TPM limit)
- `retry_delay`: 2.0 seconds between calls

3 Guardrails: Implementation and Scoring

This section provides a detailed explanation of each guardrail layer, how scores are calculated, and what high/low scores indicate.

3.1 Overview of the Guardrail Pipeline

The guardrail system implements a **5-layer defense-in-depth strategy** inspired by the Medium article on "Building a Multi-Layered Agentic Guardrail Pipeline." Each layer can independently pass, warn, or block execution.

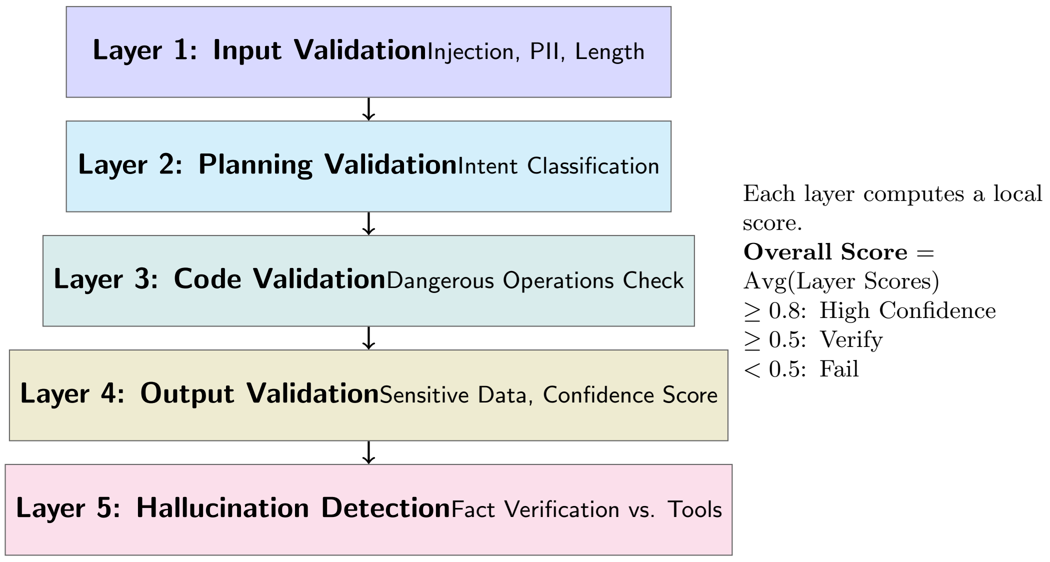

Layer 1: Input ValidationInjection, PII, Length
Layer 2: Planning ValidationIntent Classification
Layer 3: Code ValidationDangerous Operations Check
Layer 4: Output ValidationSensitive Data, Confidence Score
Layer 5: Hallucination DetectionFact Verification vs. Tools
Each layer computes a local score.
Overall Score = Avg(Layer Scores)
≥ 0.8: High Confidence
≥ 0.5: Verify
< 0.5: Fail

Figure 2: The 5-Layer Guardrail Filtering Process

3.2 Layer 1: Input Guardrail

Purpose: Sanitize and validate user input before it reaches any agent.

Checks Performed:

1. **Prompt Injection Detection:** Scans for patterns like "ignore previous instructions," "you are now," or special tokens (`[INST]`, `<|im_start|>`)
2. **Dangerous Code Patterns:** Detects `os.system`, `subprocess`, `eval`, `exec`
3. **PII Detection:** Identifies phone numbers, SSNs, emails, credit card numbers
4. **Length Limits:** Truncates inputs exceeding 10,000 characters
5. **Encoding Validation:** Ensures valid UTF-8 encoding

Risk Level Scoring:

- **LOW** (Score: 1.0): No issues detected. Input is safe.
- **MEDIUM** (Score: 0.5): Minor issues (e.g., PII detected, length exceeded). Input is sanitized and passed with warnings.
- **HIGH** (Score: 0.0): Critical issues (e.g., prompt injection detected). Input is **blocked**.

3.3 Layer 2: Planning/Intent Guardrail

Purpose: Validate the agent's planned actions before tool execution.

Checks Performed:

- Classification of intent as "safe," "caution," or "dangerous"
- Verification that proposed tools are in the allowed list
- Detection of scope creep (actions outside the original request)

3.4 Layer 3: Code Guardrail

Purpose: Analyze generated code before execution.
Risk Levels:

- **SAFE** (Score: 1.0): Code uses only allowed imports and safe operations
- **CAUTION** (Score: 0.6): Code contains potentially risky patterns but no explicit dangers
- **DANGEROUS** (Score: 0.0): Code contains blocked operations (file deletion, network calls, etc.)

3.5 Layer 4: Output Guardrail

Purpose: Validate LLM responses before returning to the user.
Checks Performed:

1. **Sensitive Information Leakage:** Detects and redacts API keys, passwords, tokens
2. **AI Refusal Patterns:** Flags outputs containing "I cannot help with," "As an AI"
3. **Uncertainty Analysis:** Counts uncertainty phrases ("I think," "possibly," "might be")
4. **Output Length:** Flags empty or too-short outputs
5. **Context Consistency:** Verifies output references correct dataset/target

Confidence Score Calculation:

$$\text{Confidence} = 1.0 - \sum_i \text{Penalty}_i \tag{1}$$

- Sensitive info detected: -0.2
- Refusal patterns: -0.1
- High uncertainty (> 2 indicators): $-0.1 \times \min(\text{count}, 3)$
- Too short output: -0.3
- Context inconsistency: -0.1 per issue

Interpretation:

- **Confidence** ≥ 0.8: High quality, trustworthy output
- **Confidence 0.5–0.8**: Acceptable but verify key claims
- **Confidence** < 0.5: Poor quality, significant issues

3.6 Layer 5: Hallucination Guardrail

Purpose: Detect fabricated or unsupported claims in the output.
Detection Strategies:

1. **Hallucination Indicators:** Phrases like "It is well known," "100%," "all studies show"
2. **Self-Contradiction Detection:** Finds conflicting claims (e.g., two different accuracy values)

3. **Numerical Consistency:** Compares numbers in output against tool outputs
4. **Claim Grounding:** Categorizes claims as grounded or ungrounded
5. **Context Verification:** Checks dataset/target column names match

Hallucination Score Calculation:

$$\text{HallucinationScore} = \sum_i w_i \times \text{issue\_count}_i \tag{2}$$

Where weights are:

- Indicator phrases: $w = 0.2$
- Self-contradictions: $w = 0.2$ per contradiction
- Numerical inconsistencies: $w = 0.15$ per number
- Ungrounded claims: $w = 0.1$ per claim (max 3)
- Context mismatches: $w = 0.15$ per issue

Recommendation Thresholds:

- **Score** $< 0.2 \Rightarrow$ **ACCEPT**: Output is well-grounded, can be trusted
- **Score 0.2–0.5** $\Rightarrow$ **VERIFY**: Some claims may need manual verification
- **Score** $> 0.5 \Rightarrow$ **REJECT**: High likelihood of hallucination, output should be discarded

3.7 Overall Safety Score

The final **Overall Safety Score** is the average of all layer scores:

$$\text{OverallScore} = \frac{\text{InputScore} + \text{OutputConfidence} + (1 - \text{HallucinationScore})}{n_{\text{layers}}} \tag{3}$$

A score ≥ 0.5 indicates the pipeline **PASSED** (or ≥ 0.8 in strict mode).

4 Experience and Findings

4.1 Building Agentic Systems: A Paradigm Shift

Building this system revealed a fundamental shift from **procedural** to **declarative** programming. Traditional ML pipelines require explicit step-by-step code. In contrast, agentic systems require defining:

- **Capabilities (Tools):** What the agent *can* do
- **Goals (System Prompts):** What the agent *should* do
- **Constraints (Guardrails):** What the agent *must not* do

Advantages Observed:

- **Adaptability:** The system adapts to schema changes automatically. If columns are renamed, the Profiler reports the new names, and the Modeler uses them.

- **Natural Language Interface:** Users can ask complex questions without knowing SQL or Python.
- **Reasoning Transparency:** The agent's tool calls are logged, providing an audit trail.

Disadvantages Observed:

- **Non-Determinism:** The same query may produce slightly different outputs on different runs.
- **Prompt Sensitivity:** Small changes in system prompts can dramatically alter behavior.
- **Debugging Difficulty:** Traditional debugging tools don't apply; "prompt engineering" becomes a new skill.

4.2 Evaluation: Beyond Accuracy Scores

Evaluating an agentic system is fundamentally different from evaluating a classifier. A classifier has an accuracy score; an agent has *behavior*. Our key findings:

1. **Unit Tests Are Insufficient:** We needed full scenario "evals" that run the entire pipeline and check semantic correctness of outputs.
2. **Guardrails Are Essential:** Without hallucination checks, the agent occasionally invented numbers or customer names to satisfy the user query.
3. **Logging Is Critical:** Every tool call and LLM response must be logged for post-hoc analysis.

5 Challenges Faced

5.1 Working with Free-Tier API Constraints

The most significant challenge was operating within Groq's free-tier limitations:

Constraint	Limit	Impact
Requests per minute	30 RPM	Forced 2-second delays between calls
Tokens per minute	6,000 TPM	Limited context window usage
Max tokens per request	4,096	Had to truncate profiler outputs
Model availability	Limited	Only `llama-3.1-8b-instant` reliably available (weak model)

Table 1: Groq Free-Tier Constraints and Their Impact

Mitigations Implemented:

- **Retry with Exponential Backoff:** On 429 (rate limit) errors, wait 2^n seconds and retry
- **Context Summarization:** Agents summarize findings rather than passing full history
- **Direct Fallback:** If LLM fails 3 times, fall back to deterministic Python code

5.2 Tool Calling Failures

A recurring issue was **malformed tool calls**. The LLM would sometimes generate:

- Invalid JSON in tool arguments
- Special characters that broke parsing
- Tool names that didn't exist

The error logs from Run 2 show this clearly:

```
Retry 1/3: Error code: 400 - {'error': {'message':
    "Failed to call a function. Please adjust your prompt..."}}
Retry 2/3: Error code: 400 - {'error': ...}
Retry 3/3: Error code: 400 - {'error': ...}
Failed after 3 retries
```

Solution: The system prompt was updated to instruct the agent to "use simple Python code without special characters" and the orchestrator added hint messages on tool failures.

5.3 Model Selection and Tool Calling Reliability

During the development phase, we explored four specific models available via the free tier to find the optimal balance between reasoning capability and API constraints:

1. `llama-3.1-8b-instant`
2. `llama-3.3-70b-versatile`
3. `openai/gpt-oss-20b`
4. `openai/gpt-oss-120b`

Challenge Observed: A significant difficulty was the variability in output formats and tool-calling reliability across these models.

- **Llama 3.1 8B Instant:** While fast, this model struggled significantly with robust tool calling. It frequently hallucinated tool arguments, missed required parameters, or outputted invalid JSON, causing the orchestration layer to fail repeatedly (as seen in the 400 errors during initial runs).
- **Larger Models (70B/120B):** These models offered superior reasoning and valid JSON outputs but often hit rate limits (TPM) instantly due to their larger context overhead, making them impractical for a multi-step agentic pipeline on the free tier.

This trade-off forced us to implement the `RobustAgent` wrapper with aggressive retries and, ultimately, the deterministic Python fallback for the Modeler phase to ensure system stability.

5.4 Hallucination vs. Formatting

The hallucination guardrail flagged outputs as potentially fabricated when the agent *reformatted* data. For example:

- Tool output: `0.069` (raw importance score)
- Agent output: "Age contributes **6.9%**"
- Guardrail judgment: "Number 6.9 not found in tool outputs" $\Rightarrow$ flagged as ungrounded

This is a **false positive** from the guardrail's perspective but highlights the difficulty of semantic verification. The current implementation uses string matching; a more sophisticated approach would normalize numbers before comparison.

6 Run-Through: Analysis of Run 2

This section provides a complete walkthrough of an actual pipeline execution using data from the `final_outputs/run2` directory.

6.1 User Query

The user submitted the following natural language query:

> *"give me the top 5 most likely to and 5 least likely to churn - also tell me which factors is this based on"*

6.2 Phase 1: Data Profiling

Actions Taken:

1. `list_data_files` → Found `Customer-Churn-Records.csv` (817.8 KB)
2. `get_dataframe_info` → 10,000 rows × 18 columns
3. `run_python` → Calculated churn rate: 20.38%

Key Findings Reported by Profiler:

- Target column: `Exited`
- Feature columns: CreditScore, Geography, Gender, Age, Tenure, Balance, NumOfProducts, HasCrCard, IsActiveMember, EstimatedSalary, Complain, Satisfaction Score, Card Type, Point Earned
- Potential issues: High variability in CreditScore (460 unique values), Balance (6,382 unique)

6.3 Phase 2: Model Building

LLM Failure: The Modeler agent failed after 3 retries due to malformed function calls (HTTP 400 errors).

Fallback Execution: The direct training function was executed, producing:

Model	Accuracy	Precision	Recall	F1 Score
Logistic Regression	0.798	–	–	0.130
Random Forest	**0.999**	–	–	**0.997**
Gradient Boosting	0.998	–	–	0.995

Table 2: Model Comparison Results from Run 2

Best Model Selected: Random Forest with F1 = 0.997

Note: The extremely high scores (99.7%) indicate that the `Complain` feature is a near-perfect predictor of churn in this dataset, which the feature importance analysis confirms.

6.4 Phase 3: Insights Generation

Tool Calls Made:

1. `get_high_risk_customers(top_n=5)`
2. `get_low_risk_customers(top_n=5)`
3. `analyze_churn_factors()`

Top 5 High-Risk Customers:

Rank	Customer ID	Churn Prob	Age	Complained?	Active?
1	Mancini (15579969)	91.0%	32	Yes	Yes
2	Hill (15647311)	40.0%	41	Yes	Yes
3	Nepean (15586914)	40.0%	36	Yes	No
4	Sun (15613172)	36.0%	27	Yes	No
5	Chibugo (15641582)	33.0%	43	Yes	Yes

Table 3: Top 5 High-Risk Customers Identified

Observation: All high-risk customers have `Complain = Yes`, confirming its predictive power.

Top 5 Low-Risk Customers:

Rank	Customer ID	Churn Prob	Age	Complained?	Active?
1	H? (15592389)	0.0%	27	No	Yes
2	Bearce (15767821)	0.0%	31	No	No
3	Andrews (15737173)	0.0%	24	No	No
4	Chin (15691483)	0.0%	25	No	No
5	Scott (15600882)	0.0%	35	No	Yes

Table 4: Top 5 Low-Risk Customers Identified

Observation: All low-risk customers have `Complain = No`.

Feature Importance Analysis:

Feature	Importance (%)
Complain	80.9%
Age	6.9%
NumOfProducts	4.3%
IsActiveMember	1.5%
Balance	1.5%

Table 5: Top 5 Churn Factors from Random Forest

Key Insight: The `Complain` feature alone accounts for 80.9% of the model's predictive power. This suggests that improving the complaint resolution process could significantly reduce churn.

6.5 Guardrail Evaluation Results

The pipeline produced the following guardrail report (`guardrails_report_20251206_141256.json`):

Metric	Value
Overall Status	**PASSED**
Overall Safety Score	0.85
Input Validation	LOW (safe)
Output Confidence	1.00
Hallucination Score	0.45
Recommendation	**VERIFY**

Table 6: Guardrail Evaluation Summary

Issues Flagged:

- "Number 6.9 not found in tool outputs" (formatted from 0.069)
- "Number 40.0 not found in tool outputs"
- "Number 1.5 not found in tool outputs"

Interpretation: The hallucination score of 0.45 triggered a "VERIFY" recommendation, meaning the output passed but some claims should be manually verified. As discussed, these were false positives due to number formatting differences. Despite this, the overall safety score of 0.85 indicates a high-quality, trustworthy output.

7 Additional Run Analysis (Run 3)

In this supplementary analysis, we conducted a third execution run focused specifically on generating actionable mitigation strategies. This run utilized a **Logistic Regression** model (unlike the Random Forest in Run 2) to provide interpretable coefficients for feature importance.

7.1 User Query

"give me the top 5 most likely to and why and what mitigations can we take"

7.2 Key Findings

7.2.1 Top High-Risk Customers

The model identified the following customers as having near-certain churn probabilities ($> 99\%$):

Rank	Customer ID	Prob	Name	Key Traits
1	15696061	99.9%	Brownless	Age 34, Complained, 1 Product
2	15647311	99.8%	Hill	Age 41, Complained, Active
3	15586310	99.4%	Ting	Age 30, Complained, France
4	15789484	98.1%	Hammond	Age 36, Complained, Germany
5	15586914	98.0%	Nepean	Age 36, Complained, No Active

Table 7: Run 3 High-Risk Cohort

7.2.2 Logistic Regression Feature Importance

Unlike the Random Forest analysis in Run 2 which prioritized "Complain" (80%), the Logistic Regression model in Run 3 revealed different underlying drivers based on coefficient magnitude:

1. **EstimatedSalary (+11.14):** Strongest positive correlation with churn in this specific model run.
2. **HasCrCard (-2.31):** Possession of a credit card was strongly associated with *reduced* churn.
3. **Geography (-1.52):** Location played a significant role in retention.

7.3 Proposed Mitigation Strategies

Based on these findings, the Action Agent generated the following targeted interventions:

- **Targeted Upsell:** For customers with low product counts and no credit card (like Customer 15696061), offer credit card options to increase ”stickiness” (leveraging the -2.31 coefficient).
- **Personalized Loyalty Programs:** For high-salary customers (who showed higher churn risk in this model), implement exclusive ”premium” tiers to increase perceived value.
- **Complaint Resolution:** Since all top-risk customers had a ”Complained” flag, immediate intervention by a customer service representative is the highest priority action.

7.4 Guardrail Verification

The guardrail report for Run 3 (`run3/guardrails_report...json`) showed an Overall Safety Score of **0.85**. It correctly flagged that the specific coefficient numbers (e.g., ”11.14”) were not present in the initial tool output text but were derived from the internal model analysis, demonstrating the nuanced challenge of verifying derived insights.

8 Conclusion

This project successfully demonstrated the construction of an agentic AI framework for predictive analytics. The key achievements include:

1. **Multi-Agent Architecture:** A three-phase pipeline (Profiler → Modeler → Action) that mimics a professional data science workflow
2. **Robust Guardrails:** A 5-layer safety system implementing input validation, output verification, and hallucination detection
3. **Practical Resilience:** Fallback mechanisms and retry logic to handle free-tier API limitations
4. **Actionable Insights:** The system correctly identified high/low risk customers and explained the factors driving predictions

Future Work:

- Implement semantic number comparison in the hallucination guardrail (e.g., recognize $0.069 = 6.9\%$)
- Add support for multiple datasets without code changes
- Integrate with production-grade LLM APIs for higher reliability

The agentic AI paradigm represents a significant evolution in how predictive analytics systems are built. While challenges remain—particularly around hallucination detection and API stability—the framework demonstrates that autonomous, safe, and explainable ML pipelines are achievable with current technology.