ChatGPT

AIVA Dashboard System Documentation

Introduction

The **AIVA Dashboard** is an AI-driven, multi-agent venture capital simulation platform. It simulates the behavior of venture capital (VC) investors through a network of AI agents, with the goal of improving startup outcome prediction over successive iterations [1]. The system combines elements of machine learning (e.g. Random Forest, XGBoost models), reinforcement learning concepts, agent-based modeling, genetic algorithms, and real-time data scraping into a unified, **Predict–Evaluate–Decide (PED)** framework [1]. By blending synthetic training data with live web-scraped startup data and multi-agent simulations, AIVA creates an evolving, feedback-driven investment analysis system [2].

The AIVA platform is designed for both local use and cloud deployment (e.g. via Streamlit Cloud). It provides a hybrid user interface built with Streamlit (for interactive controls and dashboards) and Gradio (for quick prediction demos) [3] [4]. This allows users to both **control** and **monitor** the simulation in real-time, and to obtain quick AI-driven predictions on custom startup inputs. The overall philosophy of AIVA is to **learn from layered feedback** – each phase of the system informs the next, so that *failures and successes continually refine the model's decision-making*. In essence, this is not a static ML model but a **recursive learning system** where *intelligence emerges through layered feedback, failures are leveraged across agents, and evolution reinforces diversity in strategies* [5].

System Architecture and Phases (PED Phases 1–4)

AIVA's architecture is organized into four sequential PED phases, each addressing a specific stage of model learning and evaluation. **Figure 1** below illustrates the pipeline, from initial training on synthetic data to real-world testing, multi-agent simulation, and failure-driven retraining:

Figure 1: Four-phase PED pipeline of the AIVA system. Each phase improves the model's robustness, with accuracy dropping when challenged by real data and recovering after simulation feedback and retraining.

Each phase corresponds to a cycle of **Predict → Evaluate → Decide (PED)** operations, progressively increasing the model's realism and performance. The phases are:

Phase 1: Synthetic Training (PED Foundation Layer)

Input: A large corpus of rule-based, semi-realistic **synthetic startup data** (e.g. features engineered to mimic real startup metrics). The dataset is clean and well-labeled by design [6].

Objective: Train an initial prediction model on this controlled data. A variety of models can be used (e.g. XGBoost, Random Forest, or neural networks), but all are trained to **predict startup outcomes** (such as successful exit vs. failure) with high accuracy [7]. The PED framework is initially encoded here: the model

Predicts a startup's outcome, *Evaluates* its confidence in that prediction, and *Decides* on an action (e.g. invest, wait, or reject) based on that confidence [8].

Output: A high-performing foundational model (often achieving ~90% accuracy on the synthetic data) [9]. This model forms the baseline for all future reasoning and will be carried into subsequent phases as the starting point for decision-making logic.

Phase 2: Real-Time Webscraped Testing

Input: Real-world startup data scraped from the web, including sources like AngelList, Crunchbase, TechCrunch, etc [10]. This data is ingested in real-time or from recent snapshots, and is typically noisy and incomplete compared to the synthetic data.

Objective: Stress-test the model on real-world data without retraining it. The model (from Phase 1) makes predictions on these raw, messy inputs. This reveals how the model copes with reality – e.g. missing values, out-of-distribution features, and genuine market noise.

Output: Typically a noticeable drop in performance (e.g. accuracy falling to ~70–75% on real data) [11]. This phase identifies the model's weaknesses and blind spots when confronted with actual startup profiles. Importantly, **Phase 2 does not train the model** – it only evaluates and logs performance on real data to pinpoint gaps. This **evaluation data (with model errors noted)** is saved for later phases, but the model parameters remain those from Phase 1.

Phase 3: Multi-Agent Simulation with Genetic Algorithms

Input: The pre-trained model (frozen from Phase 1) and a mix of **simulated startup market scenarios**. These scenarios can combine synthetic startups and real-data-injected scenarios to create a simulated market in which agents operate.

Mechanics: In this phase, multiple **VC agent instances** are created, all using the current prediction model as their decision-making brain. Each **agent independently runs the PED loop** – for each simulated startup, it predicts the outcome, evaluates confidence, and decides whether to invest, wait, or reject [12]. Agents act in parallel on shared deal flow and **log their actions, confidence levels, and results** for each decision [13]. Crucially, agents aren't isolated: they **share experiences and learn heuristics from one another via a trust-weighted peer network** [14]. This means agents form a graph of "colleagues" where trust scores influence how much they heed each other's successful strategies (an agent whose decisions often yield good results gains trust and influences others).

Genetic Algorithm Evolution: After a round of simulation, the system evaluates each agent's performance using a **fitness function** that can factor in metrics like ROI (return on investment of their decisions), regret (missed opportunities or bad investments), trust earned, and decision entropy [15]. The top-performing agents are then selected as "parents" and a new generation of agents is evolved by applying random mutations and crossovers to their internal decision parameters [16]. For example, an agent's risk tolerance or decision threshold might slightly mutate, producing variation. Poorly performing agents are replaced by the new offspring, creating a next generation of agents that (hopefully) perform better.

Output: Through several generations of this **simulated natural selection**, the collective performance of agents improves. We often observe the model's effective accuracy in simulation climb back toward ~80–85% [17], despite the complexity of the environment. This improvement is an *emergent result* of agents adapting strategies and the model being used in diverse scenarios. All agent decisions and outcomes are logged, including indicators of **regret** (e.g. if an agent did not invest in a startup that later turned successful, that "regret delta" is recorded) and interactions with peers.

Phase 4: Model Retraining on Failures (Feedback Refinement)

Input: The aggregated **logs of all failures and key outcomes** from Phase 2 and Phase 3. This includes: all startup features encountered, each agent's decisions and confidence levels, the actual outcomes of those startups, any regret signals (difference between expected outcome and actual outcome), the original model's prediction errors, and even context like peer influence or co-investor decisions [18].

Objective: Retrain the core prediction model using all these accumulated lessons. Rather than retraining the agents (the agents remain simple shells that use the model), we update the model itself. The idea is to teach the model to avoid both its own prediction errors *and* the common mistakes made by agents. Essentially, the model "learns from failure" – for example, if many agents consistently missed a certain pattern that led to regret, the new training data will include those missed examples (possibly oversampled or specially labeled) so the model can adjust [19]. The retraining might involve augmenting the original training data with these failure cases, weighting them more heavily, or adding new feature information that agents observed (such as trend features or peer signals).

Output: A **refined prediction model** that is more robust and accurate on real-world data, expected to reach roughly 90–92% accuracy after retraining [20]. This final model has essentially gone through a "bootcamp" of real-world challenges and agent-driven stress tests, and it forms the end product of the AIVA pipeline. In practice, this Phase 4 model can be used directly for predictions, or the entire cycle can be repeated continuously (with Phase 2 fetching new real data, Phase 3 simulating new scenarios, etc.) to keep improving the system.

Central Role of the PED Framework

Throughout all phases, the **Predict–Evaluate–Decide (PED)** framework acts as the cognitive engine of the system. Rather than being a single module, PED logic is embedded in every layer of AIVA [21] [22]:

- In **Phase 1**, the PED loop is encoded in the model's training: it learns how to predict outcomes, measure confidence (evaluation), and implicitly a decision threshold for investing vs. not investing.
- In **Phase 2**, the model executes the PED cycle on unseen real data and *logs where it is unsure or makes mistakes (failed predictions)*. These weaknesses inform the evaluation of the model.
- In **Phase 3**, each **agent** literally runs a PED loop for each decision, and these decisions (with their evaluations and outcomes) become behavior logs for analysis.
- In **Phase 4**, the model's internal PED logic is **refined** by learning from all these logged predictions and decisions, correcting its prior errors and biases [22].

This layered PED design is highly modular. The system is built so that developers can **swap in different algorithms at each PED step** – for example, you can replace the prediction model (Predict step) with a different ML model, plug in new evaluation metrics (Evaluate step, e.g. entropy or Bayesian confidence intervals), or adjust the decision policy (Decide step, e.g. simulate a more risk-averse investor) [23]. These extension points are discussed more in the Developer Guidelines section.

Technical Stack and Requirements

Programming Language: AIVA is implemented in **Python 3**, leveraging its rich ecosystem for data science, machine learning, and web development.

Core Machine Learning Libraries: The platform supports multiple ML models and libraries. In particular, it uses **scikit-learn** for baseline models (e.g. RandomForestClassifier) and can integrate gradient boosted tree models like **XGBoost**, **LightGBM**, and **CatBoost** for improved tabular prediction performance. Deep learning frameworks such as **TensorFlow/Keras** and **PyTorch** are also installed, allowing experimentation with neural network models if desired [24] [25]. This flexibility means the prediction component in the PED cycle can be anything from a simple random forest to a complex neural net.

Data Handling and Analysis: Standard libraries like **NumPy** and **Pandas** are used for data manipulation (handling feature arrays, DataFrame operations) [26]. **SciPy** is used for any scientific computations and statistics needed (e.g. in generating synthetic data or evaluating results).

Synthetic Data Generation: Synthetic startup datasets are generated using utilities like scikit-learn's `make_classification` (for creating labeled data with controllable feature informative-ness, class separability, etc.), as well as custom logic to mimic realistic distributions. This is encapsulated in functions such as `generate_synthetic_data()` [27] which uses `make_classification` to produce a dataframe of features and a binary label. The user can adjust parameters like number of features or class separability via the UI sliders (see UI Controls).

Real Data Ingestion (Web Scraping): To incorporate real-world data (Phase 2), the platform includes tools for web scraping and API calls. It uses **Requests** and **BeautifulSoup4** for HTML scraping, and **Selenium** (with **undetected-chromedriver**) for more complex pages that require running a browser [28] [29]. For example, scraping a site like AngelList might require Selenium to handle infinite scroll or login, whereas Crunchbase data could be fetched via their API or a simple GET request if available. The presence of these libraries means that the system can automatically fetch the latest startup data, though using them might require API keys or account credentials depending on the source (e.g. Crunchbase API). The scraping component is modular – one can plug in new data sources by adding new scraper scripts that output data in the expected format.

Agent-Based Simulation: Multi-agent simulations are powered by the **Mesa** framework (version 1.2.1) [30], a Python library for agent-based modeling. Mesa provides structures for defining Agent classes and a Model (environment) in which agents operate, as well as utilities for running multiple steps ("ticks") of simulation. In AIVA, Mesa can be used to manage the VC agents in Phase 3, though the current implementation simplifies the simulation logic internally (using custom loops for evolution as a "toy" model). The Genetic Algorithm for evolving agent behavior uses **DEAP** (Distributed Evolutionary Algorithms in Python) [30] for selection, crossover, and mutation operations, although a custom GA implementation is also integrated for fine-grained control. The simulation trusts and networks are handled using **NetworkX**, a graph library used to maintain trust-weighted peer graphs between agents [31].

Probabilistic Modeling (Optional): The tech stack also includes libraries like **PyMC** and **Pyro** (probabilistic programming frameworks) [32]. These allow for advanced features such as Bayesian confidence evaluation or uncertainty modeling if needed. For instance, one could use PyMC to calibrate the model's confidence scores or to perform Bayesian updates to predictions as new information comes in (this was noted as a potential enhancement and is available for researchers to experiment with).

Natural Language and External APIs (Optional): AIVA's environment has **spaCy** and **NLTK** for natural language processing, which could be used to analyze text fields (like startup descriptions or founder resumes) if integrated. It also includes **OpenAI's API** and **Hugging Face Transformers** libraries [33], suggesting that the platform can integrate large language models for tasks like parsing news about startups, generating synthetic data via GPT, or performing semantic analysis. These are not core to the PED pipeline but provide avenues for extending the platform (for example, using an LLM to estimate a startup's hype score from news articles).

User Interface: The interface is a hybrid of **Streamlit** (for the main dashboard) and **Gradio** (for an interactive prediction panel). Streamlit (v1.46.0) provides the web app framework for layouts, widgets, and live updating of results [34]. Gradio (v5.34) is used to create a small sub-app for quick predictions with a friendly UI [35]. Visualizations in the dashboard are done with **Plotly** (for interactive charts) and optionally **Matplotlib/Seaborn** for static plots [36] [37]. For example, Plotly is used to render the confusion matrix heatmap and the agent generation fitness line chart in the dashboard.

Utilities and Logging: The project uses **loguru** for logging (which provides convenient logging with thread-safe, colorful logs) [38]. **tqdm** is used for progress bars in console (e.g., showing training progress in CLI). **PyYAML** is included for configuration management (loading/saving config files), allowing the platform to use external YAML files to define simulation or model parameters if desired. The platform likely also uses JSON for data interchange (especially in the Gradio predictor, where input is given as JSON).

Environment Requirements: To run AIVA, you should have Python 3.x and pip installed. All required Python packages are listed in the `requirements.txt` (which covers the libraries above). If using the web scraping features, a Chrome browser (for Selenium) may be needed; the `undetected-chromedriver` package will try to manage the WebDriver automatically, but you may need to ensure a compatible Chrome version is present for Selenium. No GPU is strictly required (unless you switch to a deep learning model), so the system can run on a typical modern laptop or cloud instance. For Streamlit Cloud deployment, the environment specified by `requirements.txt` will be installed automatically – just note that some features like Selenium may not function in a restricted cloud environment.

Usage Instructions (Local and Cloud Deployment)

The AIVA dashboard can be used both locally (for development or private analysis) and on Streamlit Cloud (for sharing the dashboard publicly or running it without local installation). Below are instructions for each scenario:

Running Locally

1. **Setup Environment:** Ensure you have **Python 3.9+** installed. Clone the AIVA project repository or obtain the code. Then install the required packages:

```
pip install -r requirements.txt
```

This will fetch all necessary libraries (as listed above). It's recommended to do this in a virtual environment or Conda environment to avoid conflicts.

1. **(Optional) Configure Data Sources:** If you plan to use live web scraping for Phase 2, you might need to configure access. For example, if using the Crunchbase API, add your API key to the appropriate config file or environment variable. If using Selenium for certain sources, ensure Google Chrome is installed and consider specifying any required WebDriver options. These configurations can typically be set in a `config.yaml` or within the code's constants (see documentation in code for web scraping settings). For a first run or demo, you can skip this step as the platform will default to using just synthetic data if real data is unavailable.

2. **Launch the Dashboard:** Navigate to the project directory and run the Streamlit app. Usually, the main file is `dashboard.py` (or `app.py`). Use the command:

```
streamlit run dashboard.py
```

Streamlit will start a local web server and open a browser window for the dashboard (often at `http://localhost:8501`). You should see the AIVA Dashboard UI loading.

1. **Interact with the UI:** On the left sidebar, adjust the **Controls** as needed (e.g. number of synthetic rows, number of features, etc.), then click **"Train / Retrain"** to train the model on synthetic data. After training (a success message will appear), you can click **"Run Agent Simulation"** to execute the multi-agent simulation. Navigate through the tabs (Overview, Metrics, Agent Sim, Gradio Panel) to explore the results and interactive predictor.

Tip: if you update any slider or retrain the model, you may want to re-run the simulation to see updated results. The Gradio predictor tab will automatically use the latest trained model.

1. **Stopping:** To stop the app, go to your terminal where Streamlit is running and press `Ctrl+C`.

Deploying to Streamlit Cloud

1. **Repository Setup:** To deploy on Streamlit Cloud, ensure your project is in a Git repository (e.g., on GitHub) with all necessary files. Include `requirements.txt` with all dependencies. Streamlit Cloud will read this to install packages. Also include the main app script (`dashboard.py`) in the repository root or specify it in the cloud settings.

2. **Initialization:** Log in to your Streamlit Cloud account and create a new app, pointing it to your repository and the `dashboard.py` (or equivalent) file. Set any required environment secrets (like API keys for web scraping or OpenAI) in the Streamlit Cloud app settings.

3. **Resource Considerations:** The app should deploy and run automatically. Note that Streamlit Cloud has some limitations – for example, it might not support running a Selenium browser for scraping

(outbound internet access is allowed, but running a Chrome instance might fail). It's recommended to disable or stub out heavy webscraping when on cloud (perhaps use cached data or only synthetic mode). The provided code is flexible and will function with purely synthetic data if real data is unavailable.

4. **Using the Cloud App:** Once deployed, the cloud app will provide a URL where others (or you) can access the dashboard in their browser. The UI usage is the same as described in local usage. All computations happen on the server; the browser just displays the results.

5. **Updates:** If you push changes to the repository (e.g., improving the model or adding a feature), Streamlit Cloud can auto-update the app. Make sure to synchronize any changes in `requirements.txt` as well.

Troubleshooting Deployment: If the app fails to deploy or certain features aren't working on Streamlit Cloud, check the app logs on the cloud dashboard. Common issues include missing dependencies (make sure they're in requirements.txt) or attempts to use unauthorized resources (like trying to launch a browser or use a port that's blocked). The AIVA app uses an internal Gradio server on a local port; on Streamlit Cloud this should work since it's within the same container, but if the Gradio panel is not displaying, you might need to adjust settings (e.g., ensure that `server_port` for Gradio is not conflicting or use `share=True` in a cloud environment). A message *"If you get a blank panel, give it a second to boot"* is shown by the app [39] – usually waiting a moment resolves it. If it remains blank on cloud, it could be a restriction of the platform.

UI Walkthrough (Streamlit + Gradio Interface)

The AIVA dashboard UI is organized into a **sidebar for controls** and a **main content area with multiple tabs**. The interface is designed to let you configure simulations, monitor results, and perform quick what-if predictions. The hybrid approach uses **Streamlit components for layout and plots**, and an embedded **Gradio app for interactive prediction**. Figure 2 shows the dashboard layout on startup:

Figure 2: AIVA Dashboard UI (Overview Tab). The left sidebar contains user controls for data and model parameters. The main area is showing the Overview tab with a snapshot of synthetic data. Tabs for Metrics, Agent Sim, and Gradio Panel are visible at the top.

- **Sidebar Controls:** On the left, you will find a panel titled **"Controls"** [40] . Here you can adjust parameters before running the simulation:
- *Synthetic rows:* number of synthetic startup samples to generate (e.g. 6000 by default).
- *Features:* number of features (attributes) each startup has (e.g. 20 by default, up to 64).
- *Class separability:* a factor (e.g. 1.6) controlling how easily separable the synthetic classes are – higher means easier prediction.
- *Random Forest trees & depth:* hyperparameters for the RandomForest model (number of trees, and max depth – 0 means unlimited depth).
- *Agents:* number of VC agents to simulate in Phase 3.
- *Generations:* number of generations for the genetic algorithm evolution in simulation.
- *Base threshold:* the initial decision threshold for agents (the probability cutoff for "invest" vs "pass") before evolution.

There are two action buttons: **“Train / Retrain”** which initiates Phase 1 (and Phase 4 retraining if pressed again after a simulation), and **“Run Agent Simulation”** which initiates Phase 3 using the current model and parameters [41]. The sidebar is always visible for adjustments; you typically *Train first, then Simulate*. You can change parameters and retrain as needed (for example, to see how a deeper vs shallow model performs, or what happens with more agents).

- **Overview Tab:** This is the first tab in the main area. It provides a quick snapshot of the **synthetic data** that was generated and used for training. You will see a small **data sample table** (the first 20 rows of the dataset) [42] and a summary of the dataset size (e.g. “Rows: 6,000, Features: 20” displayed) [43]. This gives context on what the model was trained on. It’s useful to inspect that the synthetic data looks reasonable (features are in the expected ranges, etc.). If needed for analysis, you could scroll this or export data from here, but primarily it’s an at-a-glance view of the current dataset.

- **Metrics Tab:** After training a model, the “Metrics” tab displays the **evaluation metrics** of that model on a validation split of the synthetic data. If you haven’t trained yet, it will prompt you to do so (“Train the model first.” message) [44]. Once a model is trained, the tab shows:

- **Key metrics cards:** Accuracy, ROC AUC, and number of samples used are shown as highlighted metrics at the top [45].
- **Confusion Matrix:** A heatmap chart visualizing the confusion matrix of predictions vs actual outcomes [46]. This helps identify if the model is skewed (e.g. many false positives vs false negatives).
- **Detailed Classification Report:** A table of precision, recall, F1-score for each class (successful vs not successful), plus macro and weighted averages [47]. This is useful to see if the model favors one class or if there’s class imbalance issues.

The metrics tab essentially confirms how well the Phase 1 model learned the synthetic data. For example, you might see ~0.85–0.90 accuracy if class_sep is moderate. If you retrained the model in Phase 4 after simulation, these metrics would update to reflect the refined model.

- **Agent Sim Tab:** This tab (labeled "Agent Sim") is dedicated to the **multi-agent simulation results**. You should run a simulation (via the sidebar button) to populate it. Before simulation, it will also prompt to train the model first [48]. After running:
- You will see a success message “Simulation complete.” and a summary line showing the **best agent’s decision threshold and the best accuracy** achieved in the final generation [49]. For example, it might say **“Best Agent Threshold: 0.509 | Best Accuracy: 0.991”**, meaning the best-performing agent decided to invest in any startup with predicted probability > 0.509, and that led to about 99.1% accuracy on the evaluation set in simulation (these numbers are on the synthetic eval subset in this toy example).
- A table of the **evolution history** appears, listing each generation’s number alongside the best threshold found, best accuracy, and mean accuracy in that generation [50]. This is essentially the output of the genetic algorithm – you can scroll this table to see how metrics improved (or if they plateaued).
- A line chart titled **“Agent Fitness (Accuracy) by Generation”** plots the best accuracy and the average accuracy for each generation of agents [51]. This visualization makes it easy to see the improvement curve over generations (e.g. maybe it jumps in early generations and then levels off).

Figure 3: Multi-Agent Simulation results in the Agent Sim tab. After running the simulation, the dashboard shows the best agent's threshold and accuracy, a table of generation-by-generation statistics, and charts (like the accuracy per generation shown) to visualize agent evolution.

Together, these elements allow you to analyze how the agent population evolved. In a more advanced scenario, you might interpret the best threshold found as the optimal confidence cut-off for investing, or examine the difference between best and mean accuracy to gauge diversity among agents. For now, with the simple threshold mutation simulation, it demonstrates the concept of evolutionary improvement.

- **Gradio Panel Tab:** The last tab, labeled "Gradio Panel" or "Interactive Predictor", embeds a **Gradio interface** for making **ad-hoc predictions** using the trained model. This is a convenient way to test the model on custom inputs:
- At the top, it provides instructions: "Paste a JSON of feature values to score." The Gradio interface presents a text box where you can input feature values in JSON format [52] [53]. For example, it might show a pre-filled example like:

```
{
  "feat_0": 0.0,
  "feat_1": 0.0,
  ...
  "feat_9": 0.0
}
```

by default (the example includes a subset of features, which you can extend to all features) [54].
- There is a **Predict** button. When you click it, the JSON is sent to the model and the model's prediction comes back. The output is shown in two fields: **Probability** (the model's predicted probability that this startup is a success) and **Decision** ("Invest" or "Pass") which is determined by comparing that probability to 0.5 or some threshold [55] [56].
- For example, you might input a startup's features and get output like "prob=0.7421" and "Invest", meaning the model thinks there's a 74% chance of success so it would invest. If you input an invalid JSON or missing fields, the interface will show an error message guiding you (the predictor function is designed to handle exceptions and respond with a message) [53].

Under the hood, this tab works by launching a Gradio app inside the Streamlit app (on a local port) and displaying it via an iframe [57]. The **Gradio panel is connected to the same model object** that you train in the Metrics tab – in fact, whenever you open this tab, the latest model is passed into the Gradio interface's queue [58]. This means you should **train or retrain the model first** (in Metrics tab) before using the predictor, otherwise it will say "Model not ready. Train in Streamlit first." as a result [59].

The Gradio tab is extremely useful for quick what-if analyses. For instance, a researcher can input various hypothetical startup profiles to see how the model responds, or a VC user could input their own company's metrics to get a rough AI evaluation. It's a simple text-based interface now, but it could be extended with sliders or form inputs for each feature in the future for ease of use.

Codebase Structure and Explanation

The project's codebase is structured to mirror the PED phases and separate concerns, making it easier for developers to extend or modify parts of the system. Below is an overview of the structure and key components:

- **Data and Initialization:**
- `generate_synthetic_data(n, n_features, class_sep)` – function that creates synthetic startup datasets using scikit-learn's data generators [27]. This is used in Phase 1 to provide training data. It is a cached function (using `@st.cache_data` in Streamlit) so that re-running with the same parameters doesn't regenerate unnecessarily.
- `load_real_data()` or similar – (in a full implementation) this would handle Phase 2 data fetching. It might include scraping routines or reading from local CSVs of scraped data. In the current code, this is likely integrated in notebooks or scripts outside the dashboard (e.g., scraping done offline and saved as `master_startup_dataset_clean.csv` as referenced in logs).
- Configuration files or sections – e.g., a `config.yaml` might list data source URLs, credentials, or toggles to turn on/off certain features (like whether to actually run Selenium). The code uses PyYAML to potentially load such configs at startup.

- **Model Training (Phase 1 & 4):**

- `init_model(model_type="RandomForest", params)` – factory function to initialize a model [60]. It currently defaults to RandomForest but can be extended to return other model classes based on `model_type`.
- `train_and_eval(model, df)` – function that splits data, trains the model, and evaluates it [61] [62]. It returns the trained model and evaluation metrics (accuracy, ROC AUC, confusion matrix, classification report) which are used to update the UI [63]. This encapsulates Phase 1 training logic and can be reused for Phase 4 retraining (called when the "Train / Retrain" button is pressed again after collecting failure cases).
- The model training code is designed to be *modular*: you can easily swap in a different classifier by changing a parameter, and retraining uses the same function so it remains consistent. The output metrics are stored in `st.session_state.last_metrics` for display.

- **Predict-Evaluate-Decide (PED) Logic:** In a conceptual sense, one might implement a `PED` class or module. In the current code, the PED cycle is implicit: the model's `predict_proba` provides the Predict & Evaluate (confidence) steps, and the Decide step is applying a threshold (0.5 or evolved threshold). A more explicit implementation could have a class:

```
class PEDDecisionEngine:
    def __init__(self, predictor, evaluator, decider):
        self.predictor = predictor  # e.g. model
        self.evaluator = evaluator  # e.g. function to compute confidence
or uncertainty
        self.decider = decider      # e.g. function to decide invest/pass
```

```
based on confidence
    def make_decision(self, startup_features):
        outcome = self.predictor.predict(startup_features)
        confidence = self.evaluator(outcome)
        return self.decider(outcome, confidence)
```

This is hinted at in the design docs, where PED could be a class-based API allowing swap-in of different predictors or decision rules [23]. In the current version, the Streamlit app directly calls the model for predictions and uses simple logic for decisions in code (threshold comparisons).

- **Multi-Agent Simulation (Phase 3):**

- **Agent Definition:** In an advanced codebase, you'd have a class like `PEDAgent` representing a VC agent. Indeed, in earlier development there is a stub for `class PEDAgent` with attributes like `id` and `dna` (parameters) [64]. The agent class would contain methods to act on a startup (apply PED and decide) and to update its trust relationships. Mesa framework would instantiate many such agents and handle environment stepping.

- **Simulation Function:** The current implementation uses a simplified function `run_agent_simulation(model, df, n_agents, threshold, generations)` [65] [66]. This function:

    - Splits the data into a portion for simulation evaluation.
    - Initializes a population of agents with random decision thresholds around the base threshold [67].
    - Evaluates all agents on the evaluation set (each agent uses the model's predicted probabilities but applies its own threshold to decide invest/pass, then accuracy is measured) [68].
    - Performs a genetic algorithm loop: select top agents, breed new thresholds (adding random noise), and replace the worst agents [69] [70].
    - After generations, returns the history and best-found threshold [71].

This is a **proxy for a full PED agent simulation**. In a full implementation, instead of just evolving a numeric threshold, agents could have a more complex "DNA" (parameters like risk tolerance, trust connections, etc.), and the fitness function would consider ROI and regret, not just accuracy. The code is structured such that one could replace `run_agent_simulation` with a more sophisticated version (for example, one that uses Mesa to simulate rounds of agents bidding on startups, or DEAP to evolve multi-parameter genomes). The current output (history of best and mean accuracies) is used to populate the dashboard. - **Logging:** During simulation, agents' actions and outcomes can be logged to a data structure (e.g., a list of failure cases). In the demonstration code, after simulation the platform could compile all cases where the model/agents were wrong or had regret. In the logs we saw references to a `failure_log` list being collected [72]. These logs would then be used in Phase 4 retraining. The code architecture could include a function `collect_failures(agents_logs) -> dataframe` that prepares the failure cases for retraining (balancing classes, adding noise, etc., similar to hard example mining which we see done in the synthetic retraining snippet).

- **Model Retraining (Phase 4):**

- In code, retraining is triggered by the same `Train / Retrain` button if pressed after a simulation. At that point, one could combine the original synthetic data with the collected failure examples from simulation to form an augmented training set. In the provided logs, we see an approach of concatenating errors with some correctly predicted samples and adding noise to them (a technique known as **hard example augmentation**) [73] [74]. For example, the code snippet creates a `boot` DataFrame of all errors and an equal number of correct cases, adds some jitter to numeric fields, and retrains the model on this mixture [75] [74]. This achieved a post-“bootcamp” accuracy of ~99.84% on the training distribution [76] (very high due to possibly overfitting the synthetic distribution, but it illustrates the method).
- The code likely has a condition to avoid retraining if there were no failure cases (or simply reinitializes and trains on synthetic data again if you press retrain without new data). For clarity, in a real scenario, Phase 4 would incorporate not just synthetic errors but also errors from real data and simulation (if available). Because this is complex to do in one Streamlit run, an alternative approach is logging those failures to disk (e.g. CSV) and then manually or offline retraining the model with them. The design allows either approach.

- The retraining code writes outputs like `master_PED_predictions_refined.csv` as seen in logs [77] [78], indicating that refined predictions were saved. This suggests an architecture where after Phase 4, the refined model’s predictions on the entire dataset are stored for analysis or further use.

- **Dashboard and UI Integration:**

- All the above components tie together in the Streamlit `dashboard.py`. The Streamlit app is essentially the orchestrator: when you hit **Train**, it calls `generate_synthetic_data`, then `train_and_eval`, then stores the model and metrics [79]. When you hit **Run Simulation**, it calls `run_agent_simulation` with current model and data [80]. The UI then reads from `st.session_state` to display results. This design means the state (model, metrics, sim results) persists as you switch tabs, thanks to Streamlit’s session state.
- The **Gradio integration** is handled by:
    - creating a `shared_queue` and launching the Gradio app in a background thread [58] [57],
    - each time the tab is loaded, the current model is placed into the queue so that the Gradio app uses the latest model [81],
    - and then an `st.components.v1.iframe` (aliased as `st_iframe`) is used to embed the Gradio interface in the Streamlit page [82].
    - The function `build_gradio_app` defines the Gradio UI (textbox for JSON input and outputs) [83] [84] and the prediction function which pulls from the queue and calls the model [85]. This architecture is clever: it decouples the Streamlit and Gradio, using a queue to safely pass the model without thread issues.
- **File Structure:** In the repository, all of this might reside in a single `dashboard.py` for simplicity (as in our snippet). However, logically one could separate it:
    - `data_utils.py` for data generation and loading,
    - `model_utils.py` for model initialization, training, and evaluation,
    - `simulation.py` for agent simulation code (and agent class definitions),
    - `app.py` (or `dashboard.py`) for the Streamlit/Gradio interface, importing the above utilities.

    - plus any scripts for scraping (not used directly in the app but to prepare real data).
- Such modularization would make it easier to test parts individually. The current codebase is small enough to keep together, but as it grows (with more complex agents or models), splitting into modules is recommended.

Overall, the codebase is structured to allow easy **experimentation and extension**. You can plug in a new model by modifying `init_model` or passing a different `model_type`. You can adjust simulation details in one place (`run_agent_simulation`). And because the UI is separate from the logic (aside from calling functions), one could even run simulations headlessly (e.g. call `train_and_eval` and `run_agent_simulation` from a research notebook) without the Streamlit interface.

Future Enhancements

While AIVA is already a powerful simulation platform, several enhancements are planned or suggested to make it even more realistic and insightful:

- **Trust-Weighted Peer Networks:** Expand the agent simulation to fully utilize **graph-based trust networks** [86]. Currently, agents share info abstractly; a future version will maintain a network where each agent's influence on another is weighted by trust scores that evolve over time. An agent that consistently makes good decisions will be trusted more; others will start mimicking its strategies. Implementing this requires tracking peer-to-peer interactions and updating trust after each round (e.g., if Agent A and B co-invest and it succeeds, they increase trust).

- **Regret Analysis:** Introduce a formal **regret metric** for decisions and incorporate it into learning. Regret here means the opportunity loss from a wrong decision (false negative or false positive). For example, not investing in a startup that turned into a unicorn would yield high regret. The system can log *regret delta* for each decision (difference between actual outcome and the agent's decision) [87] and use it to adjust strategies. Future models could be trained to minimize regret, not just maximize accuracy. This also ties into dynamic threshold adjustment: if regret for missed investments is high, agents might lower their invest threshold slightly in future generations. By analyzing regret patterns, the platform could produce recommendations like "You are too conservative; you missed 3 big wins."

- **Trend Tracking:** Implement mechanisms for agents (or the model) to detect and adapt to **market trends** [86]. In the real world, startup success odds change with trends (e.g., AI startups might be hot this year, biotech next year). A future AIVA could track temporal or sector-based trends by feeding in time-stamped data and giving agents some memory or state about recent successes. Agents might specialize in certain sectors and share trend signals. This could be achieved by adding trend features to data (e.g., a moving average of success rate in a sector) or by having agents periodically retrain or recalibrate on a sliding window of data.

- **Explainable Logging and Insights:** Enhance the logging to produce **rich, explainable insights** for each decision [88]. This means every prediction and investment decision could be accompanied by a "decision trace" – what factors most influenced the prediction (feature importance for that instance), what the confidence was, whether any peers also invested (co-investor context), etc. Logging this in an easily parseable format (JSON or a database) would allow generating explanations: e.g., "Agent 5 invested because the model gave 90% confidence (key features: revenue growth and team size) and a trusted peer Agent 2 also signaled positive." Such explainable AI features would increase user trust and help debug the model's behavior. They can be surfaced in the UI, perhaps as tooltips or a dedicated "Decision Log" tab.

- **Sharpe Ratio and Risk-Adjusted Metrics:** Borrowing concepts from finance, implement **Sharpe ratio tracking** for agent portfolios [89]. Instead of just looking at raw accuracy or ROI, we evaluate agents on risk-adjusted returns. The Sharpe ratio would consider the volatility of an agent's outcomes: an agent that gets high returns but with high variance is riskier than one with slightly lower returns but consistent outcomes. By calculating such metrics, the simulation can favor agents that make not just profitable decisions, but *stable* decisions. This could be integrated into the fitness function: e.g. fitness = ROI – λ * volatility. In practice, each agent's sequence of investments (profits/losses) would be used to compute a Sharpe or Sortino ratio, influencing genetic selection.

- **Bayesian Confidence Evaluation:** (Related to regret and explainability) Incorporate Bayesian or probabilistic models to evaluate prediction confidence more rigorously [86]. For instance, using Bayesian neural networks or ensemble methods to get a distribution of outcomes. This would allow decisions to factor in uncertainty (perhaps investing only if probability > 0.8 with low uncertainty, or if uncertain, an agent might "wait" for more info). Some tools for this, like PyMC or Pyro, are already in the stack.

- **Genetic Algorithm Enhancements:** Introduce a more advanced **Genetic Trainer** as mentioned in the design notes [90]. Instead of evolving just one parameter (the threshold), have a genome for each agent encoding multiple PED parameters: e.g. threshold, a risk-aversion factor for how confidence translates to decisions, a trust coefficient for peer influence, etc. Use GA to evolve those. This would make agents more heterogeneous and potentially more robust as a group. Over many generations, we could see emergent types of investors (some very aggressive, some cautious) and measure which strategies perform best.

- **Modular Interface Modes:** Right now, Streamlit and Gradio are combined. In future, these could be separated into dedicated interfaces: e.g. a full **dashboard mode** (Streamlit) for deep analysis and a lightweight **demo mode** (Gradio or even a CLI) for quick predictions. The codebase already supports modular UI components [91], so deploying a stripped-down predictor (Gradio-only) versus the full simulation studio could be done through flags.

- **Continuous Learning & Online Deployment:** Looking further ahead, one could turn AIVA into a live system that continuously scrapes new data (Phase 2 on a schedule), retrains the model periodically (Phase 4), and perhaps even simulates agents continuously as a form of scenario planning. This essentially would make it an always-up-to-date AI VC analyst. To do this, careful automation and scheduling would be needed, plus robust monitoring to ensure data quality.

Each of these enhancements aims to make the simulation more realistic or the insights more actionable. They are non-trivial to implement, but thanks to the modular design of AIVA, developers and researchers can tackle them one by one – adding new modules or tweaking fitness functions without rewriting the entire system.

Developer Guidelines

Developers and contributors to AIVA should keep the following guidelines in mind to maintain a clean, extensible, and robust codebase:

- **Modularize by Functionality:** Follow the structure of separating concerns (data generation, model training, simulation, UI). When adding a new feature, try to encapsulate it in a function or class rather than intermixing with unrelated code. For example, if adding a new data source (say, PitchBook scraping), create a new module for it and a function `load_pitchbook_data()` instead of clogging the main app logic. This makes debugging and testing easier.

- **Adhere to the PED API Concept:** If extending or modifying the core decision logic, consider implementing it as methods of a **PED class or interface**. The design goal is that one could plug in different predictors, evaluators, or deciders without changing the surrounding pipeline [23]. For instance, you might implement `MyNewModelPredictor` with a `.predict()` method and use it in place of the default model. Or add a new evaluation metric (e.g. a calibration step) by extending an Evaluator class. Ensure any new PED component you add has a clear interface (inputs and outputs well-defined) so that it can be swapped easily.

- **Extension Points:** Take advantage of existing hooks in the system for extending functionality:

- *Model Swapping:* The `init_model` function and the `model_type` parameter are there to let you try different models. Extend that rather than writing a completely separate training routine. This ensures the new model will still work with the UI and simulation flows.
- *Simulation Fitness:* If you want to change how agents are evaluated, you can modify the fitness calculation in `run_agent_simulation` or in the Agent class. E.g., to add Sharpe ratio to fitness, incorporate it when computing `scores`. Use the logs available (the agent's decisions and outcomes) to derive new fitness metrics.
- *Agent Behaviors:* You can create more complex agent logic by expanding the `PEDAgent` class (if using Mesa) or by adding complexity to what is being mutated in the GA. Always test new behaviors in isolation if possible (e.g., create a small script where an agent processes a single deal) before integrating into the full simulation.

- *UI Components:* Streamlit makes it straightforward to add new interface elements. If you develop a new visualization or control (say, a toggle to turn on/off using real data, or a plot for regret over time), integrate it into the appropriate tab or sidebar section. Keep the UI responsive – heavy computations should be done in the back-end functions, not in the middle of `st.button` callbacks, to maintain snappy interface updates.

- **Coding Style and Documentation:** Follow Python best practices (PEP8 style). Use docstrings for all functions and classes explaining their purpose and parameters. This is especially important as this project is likely to be used by researchers – clear documentation helps others understand your contributions. Complex sections of code (like the genetic algorithm operations, or the data augmentation in retraining) should have in-line comments explaining the logic.

- **Logging and Debugging:** Utilize the `loguru` logger for debug output instead of print statements. The dashboard could be run in a console to see these logs. For example, when adding a new scraping function, log how many records were fetched or any errors encountered. When writing evolutionary code, consider logging summary stats each generation (the dashboard currently shows these in UI, but logging can help in headless mode or troubleshooting). If a new module is added (like trend detection), log the outcomes of that module (e.g., "Detected top 3 trending sectors: ...").

- **Performance and Caching:** Be mindful of performance. The Streamlit app uses `@st.cache` decorators on expensive functions like data generation and model initialization [92] [60]. If you add something that takes time (like a long scraping or a complex calculation), consider caching its result or running it asynchronously. The find_free_port and Gradio launch are examples of handling things asynchronously to not block the main thread [57]. For any long-running training (say you integrate a deep learning model), you might want to provide progress feedback (using `st.progress` or simply logging to console).

- **Testing Changes:** Whenever adding a new feature, test each phase of the pipeline to ensure nothing breaks:

- Synthetic training still works and metrics display.
- (If applicable) Real data ingestion doesn't crash the model (if your new data has different schema or missing values, handle those).
- Simulation runs to completion and returns results in expected format.
- Retraining uses the results properly and improves or at least changes the model.
- The UI updates accordingly. Use Streamlit's ability to rerun or clear cache if needed to test fresh scenarios.

Writing unit tests for pure functions (like data processing or any math calculations) is encouraged. For agent simulation, one can write tests to ensure the genetic selection is working (e.g., known population input yields expected elite selection).

- **Collaboration and Version Control:** If multiple developers are working, be careful with Streamlit's stateful nature. A good practice is to encapsulate state changes (e.g., updating session_state) in one place, or clearly document when a function relies on global state. This avoids confusion when modifying app logic. Use branches for big features and merge after testing. The repository should ideally have CI to run basic tests (if set up).

By following these guidelines, you help maintain AIVA as a **research-friendly and extensible platform**. The modular design is meant to allow swapping components without rewriting the entire system – keep that philosophy as you extend it. The PED framework, in particular, should remain an abstraction that can have different implementations plugged in. Think of AIVA as a testbed for ideas at the intersection of ML and agent simulation: keep it flexible and others (and your future self) will be able to experiment rapidly.

Audience and Use Cases

AIVA serves a broad audience interested in the convergence of AI and venture investing. Here are some key user groups and how they might use the platform:

- **Machine Learning Researchers:** For researchers, AIVA is a rich environment to study *online learning, reinforcement learning, and multi-agent systems*. They can use it to benchmark new algorithms (e.g., try a new genetic algorithm variant, or a novel confidence calibration method) and immediately see the impact on an investment simulation. Researchers interested in **transfer learning** can observe how a model trained in a synthetic domain adapts to real data. Those in **explainable AI** can use the logs to analyze decision-making processes. Additionally, the PED structure could inspire research on how sequential decision frameworks can be improved by layered feedback. The platform's code can be extended to run experiments, and because it includes real-world data aspects, it grounds theoretical research in a practical scenario.

- **Venture Capital Professionals:** AIVA can be an educational and decision-support tool for VCs or angel investors. While it's not going to replace human judgment, it can simulate "what-if" scenarios. For example, a VC could input parameters to mirror the current market and let the simulation run to see what kind of startups an AI would pick. They can examine why those picks were made (via explainable logs) and perhaps discover non-intuitive factors that correlate with success. Over time, AIVA could track market shifts – a VC could run it quarterly with updated data to see how the AI's decisions change. Additionally, as trust networks are implemented, a VC can simulate how a syndicate (group of investors) might behave – useful for understanding dynamics like investment herding or the value of having a trusted lead investor.

- **Students and Educators:** For students in AI or finance, AIVA is a fantastic **learning tool**. It provides a hands-on way to see concepts in action: supervised learning (Phase 1), evaluation (Phase 2), evolutionary algorithms and agent-based modeling (Phase 3), and model fine-tuning (Phase 4). An instructor could use it in class to demonstrate how feeding a model noisy data causes performance to drop, and how iterative learning can recover it. Students can tweak settings and immediately visualize effects, which makes abstract concepts tangible. It's also a great project base for students: they could be tasked to implement one of the future enhancements (say, add regret tracking and show its effect) – the scaffolding is there to support such projects.

- **Builders and Startup Founders:** Tech enthusiasts or startup founders interested in building their own AI-driven analysis tools can use AIVA as a **starter kit**. If someone wants to create an AI system to evaluate business plans or to simulate economic scenarios, they can fork AIVA and modify the domain. For instance, a founder could adapt the simulation to another domain (like simulating **hedge fund trading** or **credit risk analysis**) by swapping out the data and interpretation – the PED loop and multi-agent feedback architecture would still apply. The platform demonstrates how to integrate various AI components (ML models + simulation + web app) which is valuable knowledge for builders. Moreover, open-source contributors who are builders can extend AIVA itself and contribute improvements, creating a community-driven tool that benefits all users.

- **Data Scientists in Venture Firms:** Increasingly, VC firms have data science teams. Those teams can use AIVA to test their proprietary models in a simulated market setting. They might plug in their own dataset of startups (perhaps with more features or internal scores) and see how an AI agent would perform investing in those. It's a safe sandbox before deploying any AI in production. They can also use it to demonstrate to partners how an AI thinks about investments, which might complement the firm's decision process (for example, confirming hunches or highlighting overlooked deals).

In summary, AIVA is both an **educational playground** and a **prototype of AI-assisted venture analysis**. Its flexibility means each user group can derive value in a different way – whether it's learning, research, decision support, or as a foundation to build something new. By catering to a wide audience, the project aims to bridge the gap between cutting-edge AI techniques and real-world financial decision making.

Troubleshooting

Despite best efforts, users may encounter issues. Here are some common problems and solutions:

- **Installation/Import Errors:** If you get errors on `pip install` or when running the app (e.g., `ModuleNotFoundError` for a library):
- Make sure all dependencies in `requirements.txt` installed correctly. If a package failed to install (sometimes XGBoost or others might fail if compilers are missing), you might need to install system dependencies (for XGBoost on some systems, installing `cmake` or using `conda install xgboost` can help).
- Ensure you're using a compatible Python version (3.9 or 3.10 is recommended). Some libraries might not support older Python versions fully.
- If using Conda, create an environment from scratch and install requirements there to avoid conflicts.

- If the error is about `mesa` or `deap` not found, double-check the spelling in requirements and reinstall – these are less common packages.

- **Streamlit App Won't Launch or Freezes:** If running `streamlit run dashboard.py` doesn't open a browser or shows an error:

- Check the terminal for any exceptions during startup. Sometimes, firewall or network issues can prevent it from opening a browser automatically – try manually going to `http://localhost:8501`.
- If the app is stuck, it could be trying to do something heavy at startup. For example, if it's loading a huge dataset by default. By design, AIVA generates data on the fly, which shouldn't freeze. But if you modified it to load real data, ensure you're not loading millions of rows into `st.dataframe` without sampling.

- If the interface loads but no data appears, ensure the Streamlit state variables are properly initialized. (In development, forgetting to initialize `st.session_state.model` or similar could cause issues – the provided code handles this [93].)

- **Simulation Runs Slow:** The default simulation with 16 agents and 8 generations is quick (a few seconds). If you increased these substantially (say 100 agents, 50 generations), it might slow down. If you notice the browser UI becoming unresponsive during simulation:

- Consider reducing agent count or generations, or optimizing the simulation code (e.g., vectorizing some operations or offloading heavy loops with NumPy).
- Streamlit might timeout if a single callback runs too long. Ensure that long simulations have some breaks or use `st.progress` to keep the UI alive. In extreme cases, run the simulation outside Streamlit (in a separate thread) and periodically fetch results.

- Check if the random number generation (for GA) is causing any hang – it shouldn't, but if extremely large, Python's RNG might slow. Use numpy's RNG (already used) which is fast in C.

- **Gradio Panel Not Showing or Blank:** When you switch to the Gradio tab, it should load within a second or two. If it stays blank or shows an error:

- Wait a moment; the app explicitly notes to give it a second to boot [39], as the Gradio server is spun up on the fly.
- If it still doesn't show, open the browser developer console (F12) to see if there's an iframe loading error. If you see an error like " refused to connect," it could mean the local port didn't open. This might happen on certain systems or cloud deployments. Check the Streamlit logs/terminal to see if Gradio threw an error on launch.
- On Streamlit Cloud, if the Gradio panel fails, it might be due to restrictions. A workaround could be to use `blocks.launch(..., share=True)` in `launch_gradio_in_thread` which generates a public sharable link, and then use that link in `st.components.iframe`. But that requires internet and may not be ideal. Locally, that shouldn't be needed.
- Ensure no other process is using the port Gradio wants. The code finds a free port starting at 7861 [94], but if for some reason it conflicts or cannot bind, you might try changing the default or range.

- If the error says something about `queue.Empty` unexpectedly, it means the predictor couldn't get the model from the queue in time. This might happen if the model wasn't trained. Always train first, then use Gradio. If you want to be extra sure, one could modify the code to disable the Gradio tab until a model is present.

- **Web Scraping Issues:** If you enable real data scraping and face problems:

- *Selenium fails:* Ensure Chrome/Chromium is installed and that the version matches the driver. The `undetected-chromedriver` usually manages downloading, but if not, you might have to manually specify a driver path. Running headless (with a flag) might be needed on servers.
- *No data returned:* The target site might have changed their HTML or blocking scrapers. You may need to update parsing logic (BeautifulSoup selectors) or use an API alternative. Always respect robots.txt and terms of service when scraping.
- *Slow scraping:* If scraping is too slow for interactive use, consider doing it offline and feeding the app a cached dataset for Phase 2. The app can be pointed to read a CSV instead of hitting the website every time.

- *API limits:* If using APIs (like Crunchbase), watch out for rate limits. The app could incorporate caching to avoid repeatedly fetching the same data.

- **Memory Usage:** The synthetic data is by default up to 20k rows x 64 features ~ which is fine (a few MB of memory). If you crank these much higher, memory could become an issue, especially on Streamlit Cloud (which might have ~1-2GB limits). If the app crashes or the container restarts, consider lowering those parameters. Also, the confusion matrix and classification report for huge data might be large – but Streamlit handles dataframes reasonably well by truncating.

- **Model Performance Concerns:** If you find the model not performing as expected (e.g., always predicting one class):

- Check the class balance in synthetic data (it's roughly 55/45 by default [95]). If you changed weights or class_sep drastically, the model might be under- or over-predicting. Tweak those parameters to see effect.
- If using a different model (say LightGBM or a neural net), ensure you adjust parameters (a neural net might need more epochs, which the current code doesn't loop over).
- Use the Metrics tab to diagnose – low recall or precision for class 1 might indicate threshold issues or class imbalance.

- For real data, if performance is very low, that's somewhat expected (phase 2 drop). The remedy is Phase 4 retraining. If after retraining it's still low, the model might be underfitting the complexities of real data. This could lead to exploring feature engineering or using a more powerful model.

- **UI Layout Issues:** If some component doesn't display correctly (e.g., a wide table or a plot):

- Streamlit has options like `use_container_width=True` which we use for dataframes [51]. If something is overflowing, wrap it in a container with horizontal scroll or limit its size.
- If the dark/light theme contrast in the Gradio panel is off (some users might prefer matching Streamlit's theme), note that you can customize Gradio themes – but that's cosmetic.
- Multi-tab synchronization: Each tab runs somewhat independently in Streamlit's execution model (they share state though). If you find that switching tabs triggers a re-run that clears state unexpectedly, ensure you used `st.session_state` properly (the provided code does that to persist model and sim results). Always update session_state rather than local variables for things that should persist across interactions.

Most of these troubleshooting tips come down to environment setup and understanding how Streamlit and the simulation code operate. If an issue arises that's not covered above, consider raising it in the project's issue tracker if you suspect a bug. The AIVA community (or your team) can then investigate. Also, reading the verbose logs (by running Streamlit with `-v` or adding logging) can provide clues for hidden issues. Since AIVA touches many domains (ML, web, UI, simulation), isolating the problem domain is half the challenge – but the modular structure helps with that.

License and Contribution Guidelines

License: The AIVA Dashboard is an open-source project released under the **MIT License**. This permissive license allows you to use, modify, and distribute the code freely, including for commercial or research purposes, provided you include the original copyright notice. (Refer to the `LICENSE` file in the repository for the full text.) By using this project, you agree that it comes with no warranty – use it at your own risk and discretion.

Contributing: Contributions are welcome and encouraged! If you have ideas for new features, enhancements, or bug fixes, please follow these guidelines to contribute:

- **Fork & Branch:** Start by forking the repository to your GitHub account. Create a new branch for your feature or fix (use a descriptive branch name like `feature-trend-tracking` or `bugfix-simulation-crash`).

- **Discuss (Optional):** If your contribution is significant (e.g., adding a new module or a major change), it's a good idea to open an issue first to discuss with the maintainers and community. This can provide feedback and ensure your work aligns with the project roadmap.

- **Coding Standards:** Follow the style of the existing code. Write clear, concise code with comments where it's not obvious. Ensure any new dependencies are added to `requirements.txt`. Try to maintain cross-platform compatibility (Windows, Linux, Mac) – especially for file paths or OS-specific functionalities.

- **Testing Your Changes:** Before submitting, test your changes thoroughly:

- Run the dashboard locally and go through all tabs to see that nothing is broken.
- If you added a new feature, add a sample or example demonstrating it (maybe update the README or docs with how to use it).
- If applicable, add unit tests for new utility functions. (The project might include a `tests/` directory for this purpose.)

- **Commit Guidelines:** Write descriptive commit messages. For example, "Add trust-weighted update mechanism in agent simulation" is much more informative than "update code". If your commit fixes an open issue, mention "Closes #issuenumber" in the message.

- **Pull Request:** Open a pull request to the main repository when ready. The PR template (if provided) might ask for a description of changes – be sure to explain *what* you changed and *why*. Link any relevant issues. If your PR is still a draft or you want early feedback, mark it as a draft.

- **Code Review:** Be responsive to feedback. Project maintainers may request changes or have suggestions. This is a normal part of the process to ensure quality and maintainability. Don't be discouraged by critique – it elevates the project.

- **Documentation:** For any new feature, update the documentation (or comments) accordingly. If you, say, implement the "trend tracking" enhancement, add a section in the docs or README so users know it exists and how to use/interpret it. Also update any relevant docstrings.

- **Community Conduct:** Interact professionally on issue trackers and discussion boards. Follow the project's Code of Conduct (often a `CODE_OF_CONDUCT.md` file) which typically asks for respectful communication. We aim to cultivate an inclusive environment – all skill levels are welcome to contribute.

By contributing to AIVA, you not only improve the tool for everyone but also learn about this exciting intersection of AI and finance. We appreciate all forms of contributions – from reporting bugs to writing code, improving documentation, or sharing use-cases. Together, we can build a more powerful and versatile AIVA platform. Happy coding and investing!

---

1 2 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 86 87 88 89 90 91

overview.pdf

file://file-3V6P3bCvkTmaEoRc1ZAu6F

3 4 27 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 60 61 62 63 65 66 67 68 69 70 71 79 80 81 82 83 84 85 92 93 94 95 dashboard.py

file://file-2H2DEvbmu9tvxKou9LyBnp

24 25 26 28 29 30 31 32 33 34 35 36 37 38 59 64 72 73 74 75 76 77 78 AIVA.pdf

file://file-GuJXuabncrvfLN7s3En7FS