DeFi Depeg Sentinel – Technical Documentation

1. System Overview

DeFi Depeg Sentinel is a research-grade machine learning pipeline for **stablecoin risk monitoring** and **DeFi protocol observability**. It continuously ingests on-chain data from stablecoin liquidity pools and identifies anomalies that could indicate **depeg events** or abnormal pool behavior. The system combines multiple anomaly detectors, forecasting models, and a tokenized incentive mechanism to provide early warnings of stablecoin risks. Its end-to-end workflow spans data collection, anomaly detection, risk forecasting, probability calibration, alert policy logic, a monitoring dashboard, and an on-chain **token reward** contract. The target use cases are real-time monitoring of stablecoin pegs (e.g. detecting when a stablecoin deviates from \$1 parity) and observing DeFi protocol health metrics, enabling researchers and operators to catch instability in liquidity pools or pricing early.

At a high level, the Sentinel processes raw DeFi metrics (like pool prices, liquidity, TWAP/oracle prices, etc.) to detect anomalies and predict short-term future risk. When a significant anomaly is detected – for example, a stablecoin price "depeg" beyond a threshold – the system triggers alerts, logs an **incident**, and can submit a summary on-chain for transparency and community reward. By integrating a tokenized reward mechanism, the system incentivizes the reporting of validated depeg signals on-chain, minting a small amount of **SentinelData (SDT)** tokens to the reporter address. In summary, DeFi Depeg Sentinel provides an automated pipeline from **on-chain data to on-chain alert**, aiming to improve stablecoin reliability through prompt detection of peg risks.

Pipeline Summary: The Sentinel pipeline can be summarized as: **Data Ingestion → Anomaly Detection (ensemble) → Risk Forecasting → Probability Calibration → Meta-Policy Decisions → Outputs (Dashboard & Smart Contract)**. The **data ingestion** component fetches real-time pool data and reference prices. Multiple **anomaly detectors** independently score the data for irregularities. A **fusion** step combines these detectors' outputs into a unified anomaly signal. **Forecast models** then use current features (including the anomaly signal) to predict the probability of a depeg event in the near future (e.g. 10 minutes or 30 minutes ahead). These raw risk probabilities are passed through a **calibration** model to correct for bias and produce well-calibrated risk estimates. A **meta-policy controller** monitors the system's outputs and data quality – deciding whether to raise alerts, take a snapshot report, or retrain models based on drift. Finally, results are visualized on a **Streamlit dashboard** for analysts, and if a high-severity event occurs, a transaction is sent to the **SentinelDataToken smart contract** to log the event on-chain and mint reward tokens.

2. Architecture and Data Flow

Figure 1: High-level architecture of Depeg Sentinel data flow. Raw DeFi data streams through anomaly detection and forecasting components, then calibrated risk feeds into a policy layer that triggers the dashboard updates and on-chain logging.

The architecture comprises modular components orchestrated in sequence with defined data handoffs and trigger points. Data flows through the system as follows[1][2]:

- **Data Sources (Ingestion):** The pipeline periodically fetches **raw input data** from DeFi protocols. This includes on-chain pool metrics like token prices, liquidity/reserves, and external oracle rates. For example, the system queries Uniswap v3 pool contracts for the current sqrtPrice and token reserves (to derive pool price), and Chainlink oracles for reference market prices[3][4]. It also

tracks pool-specific metrics such as Time-Weighted Average Price (TWAP) gaps, Total Value Locked changes, and price spreads between DEX price and oracle price. These raw features (e.g. price deviation from \$1, recent volatility, liquidity outflow rate) form the input vector for detection and forecasting.

- **Anomaly Detectors (Ensemble):** The raw features stream into an ensemble of **unsupervised anomaly detectors** running in parallel. The system implements six detectors: Isolation Forest (IF), Local Outlier Factor (LOF), One-Class SVM (OCSVM), Cumulative Sum (CUSUM) change detection, Autoencoder (AE), and a sequential LSTM Autoencoder (AE-Seq). Each detector analyzes recent data and produces an **anomaly score** `z_i` indicating how abnormal the current state is according to that method. For example, Isolation Forest randomly partitions feature space and assigns an anomaly score based on tree path lengths (shorter = more isolated/outlier), LOF computes the local density of the point vs. neighbors (ratio > 1 indicates outlier), and OCSVM projects data into a high-dimensional space trying to exclude outliers beyond a learned boundary. The Autoencoder attempts to reconstruct the input features; a high reconstruction error indicates an anomaly. CUSUM monitors the cumulative deviation of the price from its historical average and triggers (`z_cusum = 1.0`) when a persistent shift is detected[5][6]. Each detector's output is normalized to a 0–1 range (with 1.0 indicating a confident anomaly). The detectors operate on the same input features but capture different characteristics of anomalies (e.g. IF and LOF detect spatial outliers, CUSUM detects temporal shifts, AE/AE-Seq detect nonlinear patterns).

- **Fusion Logic:** The ensemble's individual scores `{z_if, z_lof, z_ocsvm, z_cusum, z_ae, z_ae_seq}` are fed into a **fusion module** to compute a single fused anomaly metric `anom_fused`. The simplest fusion mode is a **static average** – taking the mean of all detector scores to smooth out idiosyncrasies. The Sentinel also supports more advanced blending: a **weighted blend** where each detector's contribution can be weighted (for instance, giving more weight to historically accurate detectors) and a **correlation-aware** fusion that down-weights detectors that are highly correlated with others in order to emphasize diverse signals. In practice, the system evaluates detector performance daily and could adjust fusion weights accordingly. For example, if Isolation Forest consistently has the highest precision-recall (as was the case on one day with PR-AUC ~0.70[7]), the fusion can assign it a larger weight. Correlation-aware fusion ensures that if two detectors always fire together (e.g. if LOF and OCSVM both respond to the same outliers), they do not dominate the fused score by double-counting the same information. The result is a normalized `anom_fused` score between 0 and 1 indicating overall anomaly level – where 0 means no detector sees an issue, and 1 means a consensus high anomaly (often corresponding to a known event). Notably, during a true depeg event, several detectors will spike, pushing `anom_fused` near 1.0 (indeed, in logged incidents `anom_fused` often reached 1.0 when an anomaly was confirmed[5][8]).

- **Forecaster Models (10m & 30m):** The fused anomaly signal and raw features feed into **forecasting models** that predict short-term risk of depeg. There are two gradient boosted tree models (using XGBoost) trained for different horizons: one for 10-minute ahead risk and one for 30-minute ahead risk. Each model ingests a feature vector that includes current state features and recent stats: e.g. `dev` (current price deviation from peg), `dev_roll_std` (rolling standard deviation of price, as a volatility measure), `tvl_outflow_rate` (rate of liquidity outflow from the pool), `spot_twap_gap_bps` (current spot-vs-TWAP price gap in basis points), `oracle_ratio` (ratio of pool price to oracle price), `anom_fused` (ensemble anomaly score), `r0_delta/r1_delta` (recent changes in reserve balances of the two tokens), and aggregate event features like `event_severity_max_24h` and `event_count_24h` (the worst severity and count of anomaly events

in this pool in the last 24h)[9][10]. Based on these inputs, the forecaster outputs a probability `p_horizon` that the pool will experience a "depeg" event within the next *horizon* minutes. Here an "event" is defined by a threshold on deviation or anomaly: specifically, the training label `y_horizon` is 1 if **within the next *horizon* minutes the absolute deviation |dev| exceeds 0.005 (0.5%)** or if the fused anomaly score exceeds 0.90 at any point[11][12]. Formally, for each time t_i, the label $y_i^{(10m)} = 1$ if $\max_{j \in [i+1, i+H_{10}]} |dev_j| \ge 0.005$ or $\max_{j \in [i+1, i+H_{10}]} anom_fused_j \ge 0.90$, where H_{10} is the number of samples in a 10-minute window[13]. If no such spike occurs, $y_i=0$. These labels capture the notion of an impending depeg or anomaly *soon after* time i. The models are trained (using XGBoost classifiers) on historical data labeled in this way, using a rolling dataset of recent samples. The **10m model** focuses on very short-term immediate risk, while the **30m model** looks slightly further out, which can be slightly easier to predict as it allows more time for an ongoing depeg to manifest. Both models produce an output score between 0 and 1 (a predicted probability).

During training, care is taken to ensure positive and negative examples exist. If the data is too calm and yields no positives at the default threshold, the pipeline automatically lowers the threshold (e.g. from 0.5% to 0.1%) or flags top-percentile anomaly scores as pseudo-events to avoid a degenerate all-zero label scenario[14]. For instance, if using 0.005 as the dev threshold produces no training positives, it tries 0.001 or uses the top 5% largest `anom_fused` values as events[15]. This ensures the forecasters always have something to learn, though it may make the training set skewed if real incidents are rare.

- **Calibration (Isotonic Regression):** The raw risk predictions from XGBoost are not guaranteed to be calibrated probabilities (tree ensembles often output biased confidence). Therefore, each forecaster's output is passed through a **probability calibration** stage. The Sentinel uses **isotonic regression** by default to map the model's raw probability p_{raw} to a calibrated probability p_{cal}[16][17]. Isotonic regression fits a non-decreasing piecewise-constant function on the validation data's predicted vs. actual outcomes, effectively fixing any systematic over- or under-confidence. In some configurations, a Platt's logistic sigmoid could be used as an alternative (though isotonic is chosen for flexibility in capturing arbitrary calibration curves)[17]. The calibration is done using hold-out data or the latest data available after model training. For example, the pipeline might reserve the latest 30% of samples as a validation set (as indicated by `_time_split_idx(df["ts"], 0.70)` in the code) and then compute calibration on that[18][19]. The result is stored as a set of calibration **bins** or a function that can adjust future predictions.

The calibration artifacts are saved to JSON (e.g. `calibration_10m.json` and `calibration_30m.json`) and include statistics of how the raw predictions map to true event frequencies[20][21]. Each bin entry contains the average raw prediction (`p_mean`) and the empirical fraction of positives (`y_mean`) among predictions in that range. Ideally, after calibration, these match closely (on perfect calibration, a bin with average score 0.5 would have ~50% positives). For instance, one 30m calibration bin had `p_mean` ≈ `0.857` and `y_mean` ≈ `0.917` for 36 samples【37†】, indicating the model was slightly underestimating risk at the high end (85.7% vs 91.7% actual) and the isotonic calibrator will adjust such inputs upward a bit. If the model outputs are extreme (or the dataset is very skewed), some bins may collapse. In one 10m run, **all 60 samples were labeled positive (n=60, positives=60)**, causing an "all-positive label" situation【36†】. In that case, isotonic regression cannot be fit (only one class present), so the calibrator falls back to an identity mapping – effectively leaving probabilities unchanged[22]. The documentation logs this by leaving the `bins` list empty and counts showing 100% positives【36†】. This highlights that if the forecaster becomes over-sensitive (outputting 1 for everything), calibration can't be meaningfully computed, and the pipeline would need adjustment (or use a different threshold for

labeling). The calibrated risk probabilities p_{cal} are the final outputs representing the likelihood of a depeg event in the given horizon.

- **Meta-Policy Controller:** Once we have calibrated risk estimates and anomaly signals, the **meta-policy layer** acts as a decision-making module on top. This component is implemented as a set of policy rules and endpoints (sometimes referred to as MCP, Meta-Controller Policy). It serves two primary purposes: (1) **Operational triggers** – deciding when to escalate an alert, capture a snapshot, or call the smart contract; and (2) **Maintenance triggers** – deciding if/when to retrain models due to data drift or schedule. The meta-policy ingests indicators such as the latest calibrated risk for each pool, whether data feeds are fresh, and whether feature distributions have drifted. One policy endpoint, **`/policy/decide`**, evaluates recent forecasts and system status to produce an alert level or action[23][24]. For example, if `feeds_fresh` is false (data is stale) it might decide no action, whereas if a certain pool's 10m risk exceeds a high threshold (say >60%), it might return a decision to flag that pool as "orange" or "red" alert. Under the hood, the system's `decide_latest` logic likely sorts recent events by severity and assigns each a **level**: e.g. *green* (normal), *yellow* (moderate anomaly), *orange* (significant anomaly), *red* (critical depeg imminent). These levels correspond to increasing severity and might map to thresholds on the calibrated probability or on anomaly magnitude. A red alert would trigger the most urgent actions (on-chain logging, notifications) whereas orange might only log a snapshot on the dashboard. The `decide_latest` function produces a DataFrame of recent incidents with a `level` column[25][26], and the meta-policy can act on the highest level present. Another endpoint **`/policy/snapshot`** returns a JSON of the latest incident snapshot (which includes an analyst note, top features, and context – described below), and **`/policy/retrain_check`** returns a gate signal for retraining. Internally, the meta-policy's retrain logic calls a drift detection routine and also enforces a periodic retrain interval.

- **Feature Drift Detection:** As part of meta-policy, the system evaluates **feature drift** to know when the models may be getting stale. The Sentinel computes both **Kolmogorov–Smirnov (KS) statistics** and **Population Stability Index (PSI)** for key features between the training data distribution and recent data[18][27]. The features considered for drift include `dev`, `dev_roll_std`, `tvl_outflow_rate`, `spot_twap_gap_bps`, `oracle_ratio`, `anom_fused` – essentially the core inputs to the forecaster[28][29]. If any feature shows a drift above threshold (e.g. KS ≥ 0.2 or PSI ≥ 0.25 by default), the drift detector flags an alert[30][31]. For example, in one recorded period the `dev` feature had KS ≈ 0.252 and PSI ≈ 4.926, clearly exceeding the thresholds (0.2 and 0.25 respectively) and thus `drift: True` with reason "feature drift"[30][32]. A high PSI indicates the distribution of a feature (like price deviation) in recent data has shifted significantly from the training set (PSI ~4.9 is extremely large, suggesting perhaps a new regime of data). When drift is detected, the meta-policy sets `should_retrain: True` and notes the reason[33][34]. Even if drift isn't detected, the policy may still schedule periodic retraining – the implementation always returns `should_retrain: True` either due to drift or a scheduled nightly retrain in the default configuration[35]. This ensures models are refreshed regularly (e.g. once per day) even if no explicit drift is caught.

- **Outputs – Dashboard & On-Chain:** Finally, the system produces outputs for end-users and for on-chain logging. The **dashboard** is a Streamlit web application that displays the live data and analysis results in real time (see Section 9). It receives updates either by reading artifact files or via API calls to the backend. The calibrated risk scores, latest anomaly values, and any identified incidents are pushed to the dashboard layer, which refreshes on a timer (configurable, e.g. every 30 seconds)[36][37]. Simultaneously, if the meta-policy determines an incident is severe (e.g. a "red" alert), the system prepares an **incident snapshot** and invokes the **smart contract**

integration. The incident snapshot is a compiled report (in Markdown/JSON) containing a timestamp, the pool affected, the anomaly and novelty scores, severity level, and additional context like top contributing features and network-wide cues (detailed in Section 7 and 8). The snapshot is saved to the artifacts and can be retrieved via the `/policy/snapshot` endpoint or viewed on the dashboard. For on-chain logging, the system will call the **`submit_data_block()` function of the SentinelDataToken contract** with the key data of the event: a hash of the data block, the anomaly score and novelty score (in basis points), and the severity level (1–5), along with a recipient address (who should receive the reward)[38][39]. This triggers the smart contract to immutably record the event and mint a token reward (see Section 10). In summary, normal or low-level anomalies result in updates to the dashboard only, whereas high-severity anomalies additionally result in an on-chain transaction and token emission.

All components are connected to work in streaming fashion. A scheduler (not shown in the diagram) orchestrates periodic tasks: e.g. fetch new data every minute, update anomaly scores (`/ml/score_zoo`), update forecasts, run policy checks, etc., so that the pipeline continuously refreshes. The **flow of triggers** often originates from data arrival or a clock tick. For instance, every new data point triggers anomaly detection; every few minutes the forecast is recomputed and the meta-policy `decide` is run to catch any level change; once a day (or if drift) the retrain routine triggers model retraining and calibration. These trigger conditions are indicated by dashed lines in the diagram (e.g. drift trigger causing retrain, or severe event triggering contract submission). In the next sections, we break down each component in detail, including mathematical definitions, parameters, and artifacts produced.

3. Anomaly Detectors

The Depeg Sentinel employs an **ensemble of six anomaly detection algorithms**, each denoted by a prefix `z_` in the data. These detectors run in parallel on the incoming feature stream and output anomaly scores that are then fused. Below is the list of detectors and their methodologies:

- **Isolation Forest (`z_if`):** An Isolation Forest is a tree-based anomaly detector that isolates observations by randomly partitioning the feature space. The idea is that anomalies (outliers) are more susceptible to isolation and thus have shorter average path lengths in the random trees. Mathematically, IF selects a random feature and split value repeatedly to form a tree; the anomaly score is `s = 2^{-E(h(x)) / c(n)}`, where $E(h(x))$ is the expected path length of point x and $c(n)$ is an adjustment for forest size. Lower path lengths yield higher scores (near 1). In this system, `z_if` is normalized to [0,1], with 1 indicating highly anomalous. The Isolation Forest is configured to handle the multi-feature input (deviation, liquidity, etc.), detecting points that lie in sparse regions of the joint distribution. In testing on historical data, Isolation Forest achieved the strongest precision-recall performance (e.g. PR-AUC ~0.76) among detectors【35†】, making it a key contributor to the ensemble's sensitivity.

- **Local Outlier Factor (`z_lof`):** LOF is a density-based anomaly detector that compares the local density of a point to the local densities of its neighbors. For each point, LOF computes a score based on the ratio of the average local reachability density of the point's k nearest neighbors to the point's own local density. A score ~1 indicates the point's density is similar to neighbors (not an outlier), while >1 means lower density than neighbors (outlier). We scale this into an anomaly score. The LOF detector is good at spotting if a data point is isolated from its nearest neighbors in feature space (for example, if a pool's deviation vs. volatility combination is unlike anything seen before, LOF will flag it). However, LOF can be sensitive to setting of k and can be swamped by clusters of anomalies. In our results, LOF had a moderate PR-AUC (around 0.30 in one

evaluation【35†】), indicating it catches some anomalies but also has false positives, hence it is included as one voter in the ensemble.

- **One-Class SVM (`z_ocsvm`):** A one-class Support Vector Machine attempts to learn the boundary of the "normal" data in the feature space and classify points outside that boundary as anomalies. We train an OCSVM on a sample of recent normal data (assuming most data is normal) with a kernel (e.g. RBF kernel) to capture nonlinear boundaries. The decision function $f(x)$ is positive for inliers and negative for outliers; we transform it into a probability-like anomaly score. Intuitively, OCSVM forms a hypersphere or hyperplane around the data; points that lie outside result in high anomaly scores. This detector can capture complex interactions between features that simpler detectors might miss. In practice, `z_ocsvm` showed intermediate performance (e.g. PR-AUC ~0.51【35†】), meaning it contributes useful signals to the fusion, especially for anomalies that are defined by combinations of features (like a mild price deviation coupled with an unusual oracle gap).

- **CUSUM Change Detector (`z_cusum`):** The Cumulative Sum (CUSUM) method is a statistical change point detector applied to the time-series aspect of the data (particularly the deviation `dev`). It monitors cumulative deviation from a target (e.g. 0 deviation) and flags when the cumulative sum exceeds a threshold, indicating a shift in the mean. The system's `z_cusum` is effectively a binary indicator that becomes 1.0 when a significant upward or downward shift in price deviation is detected. For example, if a stablecoin's price starts drifting downward consistently, CUSUM will accumulate that error and eventually signal an alarm. In the logs, `z_cusum` was often either 0.0 or 1.0, and during known anomalies it was 1.0 for an extended period[5][40]. CUSUM has no notion of outliers in multi-dimensional space; it purely looks at a change in the mean of a specific metric over time. It's very sensitive (often triggering on any noticeable trend), which can yield many positives (hence in one evaluation it had PR-AUC ~0.16【35†】, relatively low due to low precision). Despite this, it is useful for ensuring the ensemble doesn't miss a slow sustained depeg (which might not be "outlier" in value but is a significant change). To mitigate false alarms, the fusion logic treats CUSUM's contribution carefully (for instance, correlation-aware fusion noted that `z_cusum` was almost always 1.0 during events and might discount it when averaging so that it doesn't overwhelm other signals).

- **Autoencoder (`z_ae`):** This is a neural network (feed-forward) autoencoder trained on historical "normal" behavior of the pool. It compresses the input features into a lower-dimensional latent space and then reconstructs them. The reconstruction error $||x - \hat{x}||$ is used as an anomaly score – large error implies the current pattern doesn't conform to past patterns. The autoencoder in Sentinel is likely a shallow network given the feature vector size (the uploaded artifacts mention PyTorch 2.5 and perhaps a small model, though specifics aren't in this doc). `z_ae` is scaled between 0 and 1 based on error magnitude. It tends to capture nonlinear relationships among features – for example, if normally whenever `dev` is high, `oracle_ratio` is also high (meaning price deviates but oracle reflects it), then a situation where `dev` is high but `oracle_ratio` is normal might fool simpler detectors but produce a reconstruction error. In testing, the plain autoencoder had relatively low standalone performance (PR-AUC ~0.18【35†】). This could be due to limited training data or the difficulty of training neural nets on shifting DeFi data. Still, it provides another independent view in the ensemble.

- **LSTM Autoencoder (`z_ae_seq`):** This sequential autoencoder (denoted `z_ae_seq`) is likely based on an LSTM (Long Short-Term Memory) network that looks at a **sequence of recent data points** (time window) rather than a single point. It tries to reconstruct the sequence or predict the next

point, capturing temporal patterns. An LSTM autoencoder is well-suited for time-series anomalies because it considers the dynamic behavior – e.g. a pattern of small oscillations vs. a pattern of a sudden spike. The anomaly score here could be the reconstruction error over the sequence or the prediction error at the next timestep. In the logs, `z_ae_seq` outputs were in the ~0.7 range during anomalies[41][42]. This suggests the LSTM AE consistently assigns a relatively high anomaly score (perhaps it's normalized such that normal behavior yields lower values like 0.5 and anomalies around 0.7+). The sequential AE helps detect anomalies that manifest over multiple minutes (like a gradually accelerating depeg) which might not be obvious from a single snapshot of features. It complements the pointwise detectors by incorporating temporal context. The LSTM AE was also integrated into the system's "score zoo" update (the code comment "[zoo] anomaly scores (incl. LSTM AE) updated" indicates it runs as part of the anomaly scoring routine[43]).

Each detector produces an output between 0 and 1 for each data timestamp, which is recorded in the live dataset (the `live_dataset.csv` contains columns for each `z_*` and the fused score). The detectors can be thought of as base learners whose outputs are later combined.

Detector Fusion: The fusion logic takes `{z_if, z_lof, z_ocsvm, z_cusum, z_ae, z_ae_seq}` and computes a composite anomaly score `anom_fused`. By default, a simple average or max could be used, but the Sentinel's design allows for more nuanced fusion modes. Three modes were mentioned:

- *Simple Average:* $anom_fused = \frac{1}{6}\sum_{i=1}^{6} z_i$. This treats all detectors equally and smooths out noise. It's easy to implement but can dilute a strong signal from an excellent detector (e.g. IF) with noise from weaker ones.

- *Weighted Blend:* $anom_fused = \sum_{i} w_i z_i$ with weights w_i that sum to 1. The weights can be set based on prior performance or domain intuition. For instance, if Isolation Forest and OCSVM are known to perform well, they might get higher weight. These weights could be static or dynamically adjusted (e.g. after each nightly evaluation of detector PR-AUC, choose weights proportional to AP up to a maximum). This approach can improve the ensemble accuracy if some detectors are consistently better.

- *Correlation-Aware Fusion:* In this mode, the contribution of detectors is adjusted if they are highly correlated with others. For example, if z_{ae} and $z_{\text{ae_seq}}$ (both autoencoder-based) often produce similar scores, the fusion might treat them as one source of information to avoid doubling their influence. One could do this via a covariance matrix of detector outputs and apply a decorrelation (like multiplying by the inverse covariance matrix or using PCA to combine correlated detectors). Another simpler approach is to check pairwise correlations over a sliding window and reduce the weight of one of two detectors that are redundant. The goal is to ensure that anom_fused reflects a consensus of **independent** signals.

In practice, the ensemble fusion is likely implemented as a custom function. The evidence from the artifact evaluation suggests that a naive fusion (perhaps an average) was initially used: on one day the fused detector had PR-AUC ~0.16, even lower than the best individual detector【35†】. This indicates that without proper weighting or handling of the always-on CUSUM, the average was diluted. Recognizing this, the design allows correlation-aware improvements to avoid such outcomes. After tuning, one would expect `anom_fused` to outperform most single detectors by combining their strengths while canceling out some false positives. Indeed, the system prints out a "winner detector" daily for reference but continues to use the fused score for final decisions[44][20]. By default, we can assume `anom_fused` is at least a weighted average where detectors like `z_if` carry more weight and `z_cusum` perhaps less, to yield a balanced signal.

Detector Parameters & Inputs: All detectors operate on a feature vector derived from the current state of a pool at time *t*. The key input features include price deviation (`dev`), volatility (`dev_roll_std`), liquidity outflows, etc., as described earlier. The unsupervised detectors are typically trained or fit on a baseline of "normal" data (e.g., an Isolation Forest might be fit on the past N days of data to establish what's normal). Some detectors require hyperparameters: for IF, number of trees (e.g. 100) and subsampling size; for LOF, number of neighbors *k* (commonly 20); for OCSVM, a kernel and a ν parameter (the fraction of outliers expected, perhaps 0.05); for CUSUM, a threshold and drift step (these might be set based on domain knowledge, e.g. trigger if cumulative deviation exceeds 0.003 over 5 minutes); for AE, network architecture (e.g. 1 hidden layer of size half the input dimension); for LSTM AE, sequence length (maybe 5 or 10 steps) and network size. While exact parameters aren't listed here, an example configuration could be: Isolation Forest with 100 estimators, LOF with k=20, OCSVM with RBF kernel $\gamma=0.5$, $\nu=0.05$, CUSUM with threshold 3σ, AE with 3 layers (input-16-8-16-output), LSTM AE with window=10 and latent size=4. These would have been tuned empirically.

Performance Benchmarks: The system evaluates each detector's precision-recall performance daily to see which is most effective[45][21]. The PR-AUC (area under precision-recall curve) metric is appropriate for anomaly detection (imbalanced labels). In one evaluation (with absolute deviation threshold 0.003 as ground truth), the Isolation Forest scored highest (PR-AUC ≈ 0.768) while OCSVM was second (≈0.51). LOF was around 0.30, and the autoencoder-based detectors and CUSUM were in the 0.16–0.18 range【35†】. The fused score initially was around 0.16 (indicating a need to refine fusion) but after adjusting weighting it should improve. These metrics are saved in `detector_pr_auc.json` along with the identified **"winner" detector of the day**[44][21]. For example, on 2025-08-28 the winner was `z_if` with PR-AUC 0.704, and a PNG chart of precision-recall curves (`detector_pr_auc.png`) was saved for review[7]. Consistently, IF tends to win, which validates giving it more influence in the ensemble. Monitoring these benchmarks allows researchers to see if any detector deteriorates (maybe due to concept drift) or if an ensemble change is needed.

In summary, the detectors provide a robust, redundant layer of defense: any true depeg should trigger at least a few of them. The combination ensures higher confidence and lower false alarm rate than any single detector alone. The outputs of all detectors plus the fused score are stored for each timestamp and can be visualized (e.g. the dashboard can plot each `z_i`). This helps to diagnose which detector fired during an incident – for instance, if an anomaly was mostly caught by the volatility-based detectors vs. the deep learning ones. After detection, the next stage uses these signals for forecasting imminent risk.

4. Forecasting Models

The forecasting component of Depeg Sentinel consists of two supervised learning models that predict the probability of a depeg event in the near future. These are horizon-specific **XGBoost classification models**: one for a 10-minute horizon (predicting risk in the next 10 minutes) and one for a 30-minute horizon. By using two horizons, the system captures both immediate short-term risk and slightly longer term risk, as they can differ (e.g., an anomaly might guarantee something bad in 30 minutes but could revert in 10 minutes).

Features (Predictor Variables): The forecasters use a rich feature set derived from the pool's recent state and historical context. The primary features include[9][10]:

- `dev`: The current price deviation of the stablecoin from its peg (e.g., if the pool price for USDC is 0.98, then dev = -0.02). This is a critical feature as a non-zero deviation indicates a potential depeg already in progress.

- `dev_roll_std`: A rolling standard deviation of the deviation (or price) over a recent window (possibly last N minutes). This measures **volatility**; a sudden increase in volatility might precede a depeg or indicate instability.
- `tvl_outflow_rate`: The rate of change of the pool's liquidity (TVL – Total Value Locked). This could be computed as the percentage decrease in TVL over the last few minutes. A high outflow rate (liquidity draining quickly) is a red flag, as users pulling liquidity can signal panic or arbitrage in a depeg scenario.
- `spot_twap_gap_bps`: The gap between the current spot price and a longer TWAP (Time-Weighted Average Price), in basis points. This indicates how sharply the price has moved from recent averages. A large positive gap (spot >> TWAP) or negative gap can indicate a recent jump – essentially an alternate view of anomaly focusing on price drift relative to average.
- `oracle_ratio`: The ratio of the DEX pool price to an external oracle price (like Chainlink). Ideally for a stablecoin pool, both should be ~1. If the pool price deviates but the oracle hasn't (or vice versa), the ratio moves away from 1. A ratio far from 1 indicates either the pool or the oracle is off – often the pool might depeg while the oracle (which is slower) hasn't caught up, giving a predictive signal.
- `anom_fused`: The fused anomaly score from the detectors at the current time. This directly injects the ensemble's judgement of "how anomalous is this situation right now." If `anom_fused` is high (close to 1), it's likely an event is brewing or underway. Including this helps the model leverage the detectors' knowledge. Essentially, the forecasting model can be thought of as learning: *if anomaly detectors are all screaming (anom_fused high), then probability of a full depeg soon is high*.
- `r0_delta` & `r1_delta`: The changes in reserve0 and reserve1 of the pool (or some related metric) over the last interval. For a Uniswap v3 or Curve pool, large changes in reserves might indicate someone swapping a huge amount (which often accompanies a price change). These deltas capture **order flow** information – e.g., a big sell of USDC for USDT will reduce USDC reserve and increase USDT reserve, and also move price. Large reserve changes can be predictive of price continuing to move (momentum) or liquidity effects.
- `event_severity_max_24h`: The maximum severity of any incident in this pool in the last 24 hours. This is pulling from the events log – if, say, an "orange" or "red" event happened recently, the pool might still be in a precarious state (aftershocks of depeg, or repeated attacks). A recent high-severity event might raise the baseline risk.
- `event_count_24h`: Number of anomaly events (perhaps above some level) in the last 24 hours for this pool. A cluster of many anomalies could indicate chronic instability. This feature introduces a memory of recent history: if a pool has been acting weird multiple times, any new anomaly might be more likely to escalate than if it was the first anomaly in days.

These features are calculated continuously and stored in the live dataset. When it's time to score or train the forecaster, the latest feature values (possibly standardized or filled for NaNs) form the input vector X. Notably, the features include both instantaneous measures (like current dev) and aggregated history (like counts in 24h), giving the model both short-term and medium-term context.

Label Construction (Ground Truth): To train these models, we need binary labels indicating whether a depeg event occurred within the horizon after a given time. As mentioned, an event is defined by thresholds on `dev` or `anom_fused`. Specifically, for the 10-minute model, the label $y^{(10m)}_t = 1$ if within 10 minutes after time t, the absolute deviation $|dev| \ge 0.005$ (i.e., price moved more than 0.5% from peg) **or** the fused anomaly ≥ 0.90[13]. If neither condition happened in that window, $y^{(10m)}_t = 0$. This effectively means we consider it a "positive" if the system would have flagged an

incident in the next 10 minutes. The 30-minute label is analogous but with a 30-minute lookahead. Formally, if we denote $\Delta t = 10\text{min}$, then:

$$ y^{(10m)}*i* = *begin{cases} 1, & *max* anom_fused_j \ge 0.90,\ 0, & \text{otherwise}. \end{cases} $$ \big(|dev_j|\big) \ge 0.005 \text{ or } \max_{j: t_j \in [t_i, t_i+10min]

This labeling function is implemented in the code (function `label_targets`) which scans from each index i to $i+horizon$ and checks the dev and fused values[46][12]. There is also logic to adjust thresholds if this yields no positives in the entire dataset: it will try a lower dev threshold (down to 0.001) and if still none, it will label the top 5% of `anom_fused` values as positives[14][47]. This ensures we have a mix of 0 and 1 labels for training; however, it can introduce some weak positives which might not correspond to true incidents. The user should be aware of this and ideally ensure enough real events are present or augment the dataset.

Model Type and Training: The forecasters use **XGBoost**, a gradient-boosted decision tree library known for high performance on tabular data. XGBoost creates an ensemble of decision trees in a stage-wise fashion, optimizing a loss (here likely binary logistic loss) with regularization. We do not have the exact hyperparameters from the snippet, but typically one might use: `max_depth=3-5` (shallow trees to prevent overfit), `n_estimators=50-100` (number of boosting rounds), `learning_rate=0.1`, and perhaps a subsample fraction (0.8) to add randomness. The training happens presumably when enough new data has accumulated or when triggered manually. The code `train_forecaster_10m()` loads the live dataset, ensures labels exist (`_ensure_labels` as discussed), selects feature columns, and fits the XGBoost model[48][9]. If scikit-learn and xgboost are available, it trains and saves the model (likely into memory or to disk). The model outputs a probability p_{raw} for each input row, which is then stored in e.g. a `forecast_10m.parquet` file as `p_10m` alongside timestamp and pool.

Training is done on the accumulated dataset of live data (there may not be a static train/test split because data keeps growing). The code suggests using 70% of data for training and 30% for hold-out evaluation/calibration[49]. This hold-out is likely the tail of the data (most recent portion) which is prudent because we want to evaluate on the latest scenario. After training the model, they immediately evaluate metrics like Average Precision (AP) and Brier score on the holdout[50][51], to track model performance over time. For example, AP might measure how well the model ranks actual events high, and Brier score measures the mean squared error of the probabilistic predictions. These metrics can be logged or displayed in the nightly report for transparency.

Output Structure: Each model produces a probability (between 0 and 1) for each pool at each timestamp when it is run. These are stored in forecast results artifacts. For instance, `forecast_10m.parquet` might have columns: `ts`, `pool`, `p_10m` (calibrated or raw?), and similarly `forecast_30m.parquet` with `p_30m`. The system merges these with the live data or makes them easily joinable by timestamp and pool. On the dashboard, these probabilities are displayed as the **risk level**. For example, the dashboard computes the latest average 10m and 30m risk across selected pools and shows them as percentages[52][53]. A "10m risk (avg)" of, say, 62% means the model is predicting a 0.62 probability of a depeg within 10 minutes for the current situation on average across the pools selected. The outputs are also ingested by the meta-policy (if a pool's risk exceeds certain thresholds, it determines alert levels as mentioned).

Hyperparameter and Retraining Considerations: The horizon-specific nature means the 30m model might capture events that the 10m model doesn't, often with slightly different feature importance. For example, the 10m model might rely more on instantaneous signals (like `anom_fused` spike) while the 30m model might also leverage slower features (`event_count_24h`, etc.), since it has to predict a bit further out. We expect the 30m model to have a bit lower precision (since further out is harder to predict) but

possibly higher recall for long-simmering issues, and vice versa for 10m. The nightly training routine (if any) would retrain these models on the latest dataset if drift or schedule triggers. Retraining ensures the models adapt to new normal conditions (for instance, if a stablecoin gradually loses liquidity over weeks, the model can learn that a smaller absolute TVL might not indicate panic if it's the new normal).

Label Imbalance and All-Positive Issue: As seen in calibration, there was a case where the model's labels were all positive (which likely means at that point every window had an event – possibly due to threshold being too low or data being too volatile). In such a scenario, training a model is meaningless (it would just predict positive always). The pipeline's safeguard `_ensure_labels` tries multiple thresholds to get at least some negatives[14]. If it still fails, one might consider using a time-based split (e.g. treat every N-th interval as a negative by assumption if truly the system thinks everything is an event). This is a tricky scenario and in practice would indicate re-evaluating the event definition or adding a human-labeled incident definition. The system log shows for 10m horizon, all 60 recent points were labeled 1 【36†】, which is problematic. The calibrator then set identity (no calibration) for that case[22]. It's likely a transient or a simulation artifact, but developers should monitor if the forecaster is essentially over-triggering. If so, adjusting the dev threshold upward or fused threshold upward for labeling might be needed (e.g., require dev > 0.5% *and* fused > 0.9, not or, to label an event, to make events rarer). The documentation of thresholds (0.5% dev, fused 0.9) is an important part of the model's assumptions.

Performance and Evaluation: The system tracks **Average Precision (AP)** and **Brier Score** for these models on holdout data[54][51]. Average Precision is particularly appropriate for the highly skewed data (many no-event vs few events). Brier score (mean squared error of probability vs outcome) is a calibration-sensitive metric (lower is better). These give insight into model quality: for example, an AP of 0.8 would indicate very good ranking of event vs no-event cases, whereas a high Brier might indicate probability outputs need calibration. Using the calibration step, Brier should improve. The nightly report likely includes a section on "Forecast Calibration" with links to calibration plots and possibly AP/Brier metrics[20][55]. Ensuring the model remains performant is part of the meta-policy – if performance drops (as hinted by "drift threshold (PR-AUC drop) hit" in retrain_check[35]), that can trigger retraining too.

In summary, the forecasting models convert the anomaly detection signals and other metrics into a forward-looking risk estimate. They are a learned model that encapsulates typical pre-depeg patterns. When they raise a high probability, it implies the current market/pool conditions resemble those preceding known depeg incidents. These probabilities then go through calibration and into the decision logic and UI.

5. Calibration of Risk Predictions

After obtaining raw risk predictions from the forecasters, Depeg Sentinel applies a calibration step to ensure these probabilities are interpretable and reliable. The calibration strategy used is **Isotonic Regression**, with a fallback to Platt's sigmoid if specified (the code supports both but defaults to isotonic)[16][17].

Why Calibration: Machine learning models, especially boosted trees, can produce scores that are not true probabilities – they might be overconfident or underconfident. Calibration aligns the model output with the actual observed frequency of events. For example, if the model says "0.8 probability" 10 times and it's calibrated, about 8 of those should actually result in events.

Isotonic Regression: This is a non-parametric calibration method that fits a non-decreasing function mapping raw probabilities to calibrated probabilities. Given a set of model outputs p_{raw} and true outcomes, it will create a stepwise-constant function (isotonic = monotonic non-decreasing) that

minimizes the difference (often using least squares on the difference between predicted and actual label in a monotonic constrained way). The advantage is it can correct any monotonic distortion without assuming a shape (unlike logistic which assumes a sigmoid shape).

In the Sentinel, after training a forecaster, they gather the raw predictions p_{raw} on a validation set and the corresponding binary outcomes (did an event happen). They then create a binary label for severity or risk. The code for severity calibrator, for instance, does `y_bin = (y >= 3).astype(int)` for severity labels[56] – in that context, they treat severity >=3 as a "positive" to calibrate severity prediction. For the risk forecasts, they likely just use the 0/1 event labels for positives. Then, if there are at least some positives and negatives (`np.unique(y_bin)` has both 0 and 1)[22], they proceed to fit the isotonic regression (`IsotonicRegression(out_of_bounds="clip")` from scikit-learn)[17]. If by chance the validation set has all positives or all negatives, they detect that and set the model to identity (no calibration)[22]. This happened in the 10m case where all 60 labels were 1, triggering the `identity` model path (meaning the calibration function is just $p_{\text{cal}} = p_{\text{raw}}$)[22] 【36†】.

Once fitted, the isotonic model can predict for new raw probabilities: `p_cal = iso.predict(p_raw)` which will give calibrated probabilities[57]. The code ensures the output is clipped to [0,1]. For logistic calibration (Platt's method), they would fit a logistic regression on p_{raw} vs y (which is essentially fitting a sigmoid function), but they only use that if `method="sigmoid"` was specified (by default they pass "isotonic").

Calibration Artifacts: The results of calibration for each horizon are saved as artifacts. Specifically, `calibration_10m.json` and `calibration_30m.json` contain the calibration curve information. The structure (from the uploaded files) is like:

```
{
  "horizon_min": 10,
  "bins": [
     { "bin": i, "p_mean": ..., "y_mean": ..., "n": count_in_bin },
     ...
  ],
  "counts": { "n": total_validation_points, "positives": number_of_events }
}
```

If the model was perfectly calibrated, we'd see `p_mean` roughly equal to `y_mean` in each bin. Each bin corresponds to a group of validation samples that had similar raw predictions. For example, in `calibration_30m.json`, we saw one bin (likely the highest bin, bin index 9) with `p_mean ≈ 0.857` and `y_mean ≈ 0.917` with `n=36` 【37†】. This means in that bin, the model on average predicted 85.7% probability, but actually 91.7% of those cases were positives. The isotonic regression likely will adjust predictions in that range upward (closer to 0.91) to fix this underestimation. If there were more bins (e.g., bins 0 through 9 for deciles of predictions), we'd see lower bins perhaps where `p_mean` might be 0.2 vs `y_mean` 0.1, etc. However, in that particular file, only one bin was populated, indicating most predictions were high (the model was almost always outputting >0.8 probabilities on the validation set). This again speaks to a highly skewed scenario.

In `calibration_10m.json`, we saw `"bins": []` and counts n=60, positives=60 【36†】. An empty bin list suggests the calibrator did nothing (identity) because it couldn't fit a curve with only one class. The code sets `self.model = ("identity", None)` in that case[22], and indeed that results in no bins to save. The system likely notes that scenario in logs and uses identity mapping for that horizon.

All-Positive Label Issue: As noted, if all examples are positive, calibration cannot be done. This is a symptom of either the model or labeling threshold being too lenient. The documentation explicitly asks

to note this “all-positive label issue and how it impacts the curve.” The impact is severe: the calibration curve is degenerate (no slope to learn), so the system cannot adjust probabilities – it will just pass them through. Moreover, an all-positive outcome means the model cannot discriminate anything (it’s a sign that effectively everything is being considered an event). In practice, if this occurs persistently, developers should revisit how events are defined or possibly incorporate a time buffer (e.g., once an event happens, skip labeling overlapping windows as separate events to create some negatives). It’s mentioned likely as a caution that in such cases isotonic regression falls back to identity.

Using Calibrated Forecasts: After calibration, the system uses the calibrated probabilities for decision-making and user display. For instance, the values shown on the dashboard for “10m risk” are presumably calibrated percentages (since they are meant to reflect true likelihood)[58]. Also, the meta-policy thresholds (like what constitutes a red alert) would ideally be based on calibrated probabilities – e.g., if they decide that >50% chance should be “red”, that makes sense only if 50% is indeed 1 out of 2 times an event, which calibration assures.

Evaluation of Calibration: The calibration can be evaluated by Brier score or by reliability diagrams. The nightly report likely outputs the calibration plot images (`calibration_10m.png`, `calibration_30m.png`) alongside the JSON data[20][55]. These plots would show predicted probability vs actual frequency. A near 45-degree line is ideal. If the line is above diagonal, model underestimates (actual freq > predicted), if below, model overestimates. The `counts` in the JSON show how many samples were used – e.g., 36 in the 30m case (possibly they used the last ~30% of data, which was 36 points, and of those 33 were positives【37†】, yielding that near all-positive scenario).

Additionally, the system prints Brier score in the `_eval_forecaster_metrics` function[54][51]. This function likely uses the calibrated model (since it mentions “current loaded (calibrated) model”[59]) to output a Brier. A lower Brier after calibration would confirm the calibration improved things.

Isotonic vs Sigmoid: The code allows logistic calibration too[17][60]. Isotonic is chosen if not specified otherwise. Logistic (Platt’s) might be more stable when data is scarce, but isotonic is more flexible at the cost of possibly overfitting if validation set is small. Given the small validation sizes (36, 60, etc.), isotonic might produce weird step functions. It might be wise to use a pooled calibration from multiple days or use some smoothing. If needed, an option is to use a **platts** method by setting `method="sigmoid"` when training calibrator, which fits a simple logistic regression of form $\sigma(a \cdot p_{\text{raw}} + b)$. The code shows how it would do that (fit `lr = LogisticRegression` on the raw pred as 1 feature)[61]. In either case, the calibrator is saved (potentially using joblib if `_JOBLIB_OK`) to disk for use at inference[62][63].

Calibration of Severity vs Risk: The system also calibrates **severity scores** (discussed in next section) using a similar approach but focusing on text-based severity predictions. Keep in mind risk calibration is about aligning model probability with actual event probability, whereas severity calibration aligns a language model’s output with actual severity categories. Both use isotonic regression under the hood.

In conclusion, calibration is an essential post-processing for the forecasters: it transforms a somewhat arbitrary risk score into a probability that can be interpreted by analysts and used in automated decision thresholds. The artifacts (JSON and PNG plots) allow verifying that, for example, when the system says “20% depeg risk”, historically ~1 in 5 of such cases did lead to a depeg event, which is crucial for trust in the system.

6. Severity Modeling

Apart from predicting the likelihood of an event, the Sentinel also evaluates the **severity** of detected anomalies. Severity is intended to capture how serious or impactful an event is, on a qualitative scale (usually 1 to 5). This can incorporate aspects like amplitude of deviation, speed of onset, and novelty of the event. In simpler terms, severity answers *"How bad is this anomaly?"* whereas the risk forecasters answer *"What's the chance this will turn into a problem soon?"*.

Severity Score Definition: The system defines severity on a 1–5 integer scale, with 1 being lowest (negligible anomaly) and 5 being highest (extreme event). The severity determination in the Sentinel is handled by a separate ML component that analyzes event context (particularly text summaries of events) to assign a calibrated severity. Notably, severity is *not* directly assigned by the anomaly detectors; instead, it appears to be derived from either descriptive features or an NLP model on the event summary. The design likely considers multiple dimensions of an anomaly: - **Magnitude (Amplitude):** How far from the norm did the metric move? (e.g., a 0.3% deviation vs a 5% deviation – latter is more severe) - **Sharpness (Rate of Change):** Did it happen suddenly or gradually? (a sudden crash might be more severe due to shock and potential for cascade) - **Novelty:** Is this type of anomaly unprecedented or very rare? (a novel failure mode might be deemed more severe because it's unexpected and unmitigated) - **Duration:** A long-lasting depeg could be more severe than a transient blip. - **Cross-market impact:** Did multiple related pools also move (which might lessen severity if it's a broad market move), or is it isolated (possibly indicating something wrong with this specific pool)?

Many of these aspects can be captured in an **analyst summary text** for the incident. For example, an auto-generated summary might say: *"USDC/USDT pool depegged by 4% within 10 minutes, hitting $0.96. Liquidity dropped 15%. Similar pools remained stable."* This summary implicitly covers amplitude (4% depeg), sharpness (10 minutes), maybe novelty (if similar pools stable, this is isolated – potentially more severe as it's specific).

The Sentinel leverages an **NLP model** to estimate severity from such text. The code base references SentenceTransformer (SBERT) models for text embeddings[64], implying that when an incident occurs, they create a textual description ("summary") of the event and feed it into a model that predicts severity.

Severity Model: While the exact architecture isn't fully detailed, one likely approach is: - Use a pre-trained language model (like SBERT) to embed the event summary text into a vector. - Possibly augment that vector with some numerical features (like max deviation, etc.) if not fully described in text. - Then use a classifier or regressor that outputs a severity score (1–5). - This could be implemented as a simple logistic regression or small neural net on top of the embedding that was trained on historical labeled events.

In the code, `_sev_predict_proba(texts)` is called to get raw probabilities for severity[65][66]. This likely means the model outputs a probability that the event is "high severity". They specifically treat severity >= 3 as the positive class for calibration[56], meaning the model might originally output a probability that severity is 3 or above (i.e., a serious incident). Alternatively, `_sev_predict_proba` might output a distribution or some score which is then interpreted.

From the severity calibration code: - They collect all events' summaries and severities from `events.json`[67]. - `y_1to5` is the array of actual severities (1–5)[65]. - `p_raw = _sev_predict_proba(texts)` gives presumably for each text a probability that the event is severe (or some model output). - They then fit `_SeverityCalibrator` on `p_raw` vs `y_sev` thresholded to >=3[56][65].

This suggests the underlying model (`SEV_MODEL`) produces a score related to severity. Perhaps `SEV_MODEL` is a fine-tuned classifier that predicts one of 5 classes or a continuous severity, but they simplify it to a binary "is it >=3 or not" for calibration. The mention of `if hasattr(SEV_MODEL.model, "prior")` suggests the model might use some prior probability or baseline[68]. Could be a zero-shot classifier or a rule-based model with a prior. It also sets `prior` to 1.0 if text is empty or none[69], meaning if no summary, they assume highest uncertainty (or default high? Actually, they set 1.0 if text is blank, which means treat it as severe by default to be safe? Possibly).

Severity Calibrator: The class `_SeverityCalibrator` is similar to the forecaster calibrator but specifically for severity labeling[70][71]. It takes `p_raw` (which one might interpret as the model's predicted probability that severity ≥3) and true labels (`y_bin` indicating whether actual severity was ≥3). It fits isotonic or logistic the same way. If all events in the calibration set are of one kind (say all severity were ≥3), it will set identity[22].

After fitting, they can use this calibrator to **predict calibrated severity probability**: `_sev_calibrator.predict_proba(p_raw)`. Then they map that probability back to a 1–5 severity level. The code does:

```
sev_prob = calib.predict_proba(p_raw)
sev = np.clip((np.floor(sev_prob * 5) + 1).astype(int), 1, 5)
``` 【22†L127-L135】.
This line takes the calibrated probability (which is between 0 and 1) and multiplies by
5, floors it, and adds 1, resulting in an integer 1–5. Essentially, it divides the [0,1]
probability range into 5 equal bins:
- 0.0–0.199... -> floor(0–0.99)=0 +1 => severity 1
- 0.2–0.399... -> floor(1–1.99)=1 +1 => severity 2
- 0.4–0.599... -> severity 3
- 0.6–0.799... -> severity 4
- 0.8–1.0   -> severity 5

This is a heuristic way to map a continuous probability to five discrete levels. Note
that it doesn't use the actual multi-class nature of severity; it's effectively saying
"if the calibrated probability of being severe (≥3) is X, then distribute that on 1–5
scale linearly." If p_cal = 0.8, severity = 5; if p_cal = 0.5, severity = 3; if p_cal =
0.1, severity = 1, etc. This is a bit simplistic because it assumes the difference
between 4 and 5 is the same probability jump as between 1 and 2. But lacking a model that
directly predicts 5 classes, this at least gives a reasonable spread. They then overwrite
each event's severity with this calibrated value in the events list 【22†L131-
L139】 【22†L141-L147】. So ultimately, whenever events are saved, their severity has been
normalized to this scheme.

Interpretation of Severity:
- **Level 1–2:** Minor anomalies. These might be false alarms or tiny deviations that
resolved. The system likely doesn't alert on these; they might not be shown prominently.
- **Level 3:** This is the threshold of significance (the calibrator used ≥3 as
positive). So severity 3 is a moderate incident. It could correspond to something like a
1–2% depeg that recovered quickly or required attention but not crisis. In some contexts,
it might be considered an "orange" alert (depending on how they map it).
- **Level 4:** Major incident. Perhaps a noticeable depeg (e.g. 5% drop) that either
recovered or took time.
- **Level 5:** Critical incident. This would be something like a full depeg (stablecoin
at $0.9 or below), or a cascade failure, or widely impacting event. It's the highest
category, probably rare.

The severity score captures not just how far the price moved (amplitude) but context: e.g., if a stablecoin moved 5% but it was due to a market-wide move where everything moved 5%, that might be less *severe* (in a relative sense) than a stablecoin moving 5% in isolation due to a hack or insolvency rumor. The latter would get a higher severity since it's an anomaly unique to that asset. The mention of "novelty" in the user's prompt likely refers to this context – if an event is **novel** (meaning the pattern of metrics hasn't been seen before), it could be more severe because it might indicate a new type of failure (thus unpredictable and dangerous). The ensemble's `novelty_bps` (passed to the contract) tries to quantify novelty. Possibly one of the detectors (like OCSVM or autoencoder) is mainly measuring novelty of pattern vs training distribution, and they package that separately.

In computing severity, novel patterns might be given higher weight. For example, if the anomaly was detected primarily by the autoencoders and not by threshold detectors, it could be something very unusual – possibly raising severity. Another factor is **sharpness**: a super sharp drop might indicate a panic, which is severe. The system likely encodes sharpness via `dev_roll_std` or by language like "within 5 minutes" in summary. If summary says "quickly" or "sudden", maybe the model picks that up.

Severity Calibration and Outputs: The severity calibration ensures that the distinction between, say, severity 2 and 3 is meaningful. They calibrate such that severity ≥3 corresponds to events historically considered significant. If, for instance, their events dataset had mostly severity 3 and few 4 or 5, the isotonic calibration will effectively ensure that only truly high raw scores get mapped to ≥3. If an event summary model was too eager to call everything severe, the calibration will lower those probabilities. The end result is the final severity stored in `events.json` is more consistent.

The `events.json` presumably contains an array of event objects, each with fields like timestamp, pool, summary (text), initial severity, and maybe novelty and anomaly scores. The pipeline updates this file with calibrated severity for each event【22†L139-L147】. The dashboard or snapshot can then display severity badges (1–5) for incidents.

Use of Severity: Severity is used by the meta-policy to decide alert levels:
- Possibly severity ≥3 corresponds to "red" alerts requiring immediate attention, whereas 2 might be "orange" and 1 not alerted. In the alert logic from `decide_latest`, they separated "red" vs "orange" incidents【30†L19-L27】【30†L21-L24】. It's likely severity ≥4 or 5 triggers red, while 3 might trigger orange (depending on how cautious they are – they might treat 3 as low red or high orange).
- The smart contract submission requires a `severity` integer. They pass this severity on-chain to characterize the data block【4†L575-L579】. The contract does not enforce meaning of 1–5 beyond using it in reward calc, but it's encoded in the event log for observers.

Reward & Severity: The severity value is fed into the reward formula on-chain: recall the contract's `_compute_reward` multiplies by `severity/5`【24†L139-L147】【24†L141-L148】. This means a severity 5 event yields a reward multiplier of 5/5 = 1 (full reward), whereas severity 1 yields 1/5 = 0.2 of the base reward. So higher severity events give more SDT tokens to the reporter. This provides an incentive to correctly classify severity (and not overstate it, since presumably the reporter is the system itself and it would be dishonest to inflate severity – but if this were community-reported, someone might try to claim an event is severity 5 to get more tokens; hence severity needs to be determined by the system, not the user input, in a reliable way).

Example: Suppose an event occurred where USDC depegged to 0.95 (-5%) within 30 minutes after a governance hack rumor. The detectors fire, forecast maybe gave ~0.9 probability (which happened), and an incident is recorded. The summary might note “USDC dropped 5% in 30m on hack rumor; unprecedented isolated event.” The severity model likely outputs a high raw score (since 5% is huge and “unprecedented isolated” indicates novelty). After calibration, maybe $p_{\text{sev_cal}} = 0.95$, which maps to severity = 5. So this incident gets severity 5. The contract call would include severity=5 and anomaly_bps (500) and novelty_bps perhaps high as well, yielding a relatively large token reward. Conversely, if an event was a 1% slip that recovered in 5 minutes, summary might be “USDT momentarily lost 1% but reverted; similar moves seen across market.” That might yield a lower severity raw score, maybe $p_{\text{sev}} = 0.3$, calibrates to ~0.25, which maps to severity 2. That event might not even trigger a contract call if policy requires ≥3, but it’s logged.

Artifacts and Meta: The severity calibration metadata is saved in `severity_calibrator_meta.json` (likely containing timestamp, method, number of events used, etc.). Indeed, the code calls `SEV_CALIB_META.write_text(json.dumps(meta))` after training the calibrator【22†L111-L118】. This meta might record the method (isotonic vs sigmoid) and how many events were in the calibration (to track reliability).

In summary, severity modeling in Sentinel is a post-detection classification that assigns a *rank* or *magnitude* to anomalies. It uses text analysis of the event’s context combined with calibration to standardize those ranks. This helps differentiate between minor blips and major crises, informing both the level of response (UI coloring, on-chain logging, etc.) and the reward distribution. By calibrating severity with historical data, the system aims for consistency: e.g., all events marked severity 4 are of roughly comparable seriousness in hindsight.

7. Explainability and Feature Attribution

Understanding **why** the model is predicting a high risk or flagging an anomaly is important for researcher trust and debugging. The Depeg Sentinel includes an **explainability** component that provides insight into which features are driving the forecaster’s predictions. Specifically, it produces feature importance metrics for the 10m and 30m risk forecasts, likely using a permutation importance or SHAP (SHapley Additive exPlanations) approach.

Features for Explanation: The features considered in explanation are the same as those used by the forecaster models (see Section 4). In the `explain.json` and `explain_30m.json` artifacts, we see references to features like `dev, dev_roll_std, tvl_outflow_rate, spot_twap_gap_bps, oracle_ratio, anom_fused, r0_delta, r1_delta, event_severity_max_24h, event_count_24h` – matching our feature list. These are the features the model looks at, so those are the ones for which contributions are explained.

Method of Explanation: The documentation snippet and files suggest they used a permutation-based feature importance measured in terms of **Average Precision (AP)**. In the `explain.json` output, each feature has a “mean” and “std” and they list top contributors as e.g. `"dev (+0.3661 ± 0.0130 AP)"` 【29†】. This indicates that the feature `dev` contributed an **increase of ~0.3661 in AP** when included, with a std of ±0.013. In other words, if you randomly shuffle (or remove) the `dev` feature in the model input, the model’s average precision drops by ~0.366 (which is huge). That implies `dev` is by far the most important feature. Meanwhile, features like `anom_fused` contributed +0.0070 AP (with std 0.0177, which is very small, possibly statistically negligible), and others had 0.0. This looks exactly like a **permutation importance**

analysis: measure performance (AP) of the model on some evaluation set normally, then permute each feature (break its relationship with the outcome) and measure performance again. The drop in performance indicates how important that feature was.

From `explain.json`:
- `"top_contributors": [ "dev (+0.3661±0.0130 AP)", "anom_fused (+0.0070±0.0177 AP)", "dev_roll_std (+0.0000±0.0000 AP)" ]` 【29†】. This implies:
  - Using all features, the model had some baseline AP (not explicitly given, but we can infer baseline AP might be ~0.366 if dev alone gives that? Actually, likely baseline AP is 0.0 if no features, then dev gives 0.366 AP? Or more likely, they computed marginal contributions by some process).
  - `dev` stands out with ~0.366 AP contribution. This aligns with intuition: how far off peg is the pool is the single biggest factor in predicting a depeg event (if you're already 3% off, chance of continuing is high).
  - `anom_fused` gave ~0.007 AP, basically negligible. That's interesting: it suggests that once `dev` is known, the additional info from the anomaly detectors is not adding much in that scenario. Possibly because `dev` alone identifies most events (the events might be defined by dev anyway), making the ensemble anomaly largely redundant.
  - All other features (volatility, outflows, etc.) had 0.0 AP contribution in that output. This might be because the particular test context had nothing much to add beyond dev, or because the model leaned almost entirely on dev. It could also mean the evaluation dataset was small and not varied enough to see effects of those features.
  - The presence of ± values suggests they repeated the permutation multiple times (8 repeats for 30m as indicated in code: `explain_forecast_30m(n_repeats=8)` 【25†L13-L18】【25†L15-L20】) to get a stable estimate and standard deviation.

Thus, the method is likely **permutation importance on AP**. This is a bit unusual (many would use SHAP or feature gain from XGBoost directly), but permutation AP is fine for measuring contribution to ranking performance. Another possibility is they used an in-built XGBoost feature importance (like gain or SHAP) but the formatting suggests permutation since SHAP typically would be in terms of contribution to log-odds or output, not AP metric. The code snippet shows `exp30 = explain_forecast_30m(n_repeats=8)` and then they print `exp30.get("top_contributors")` 【25†L15-L18】【25†L99-L102】. The explain_forecast function presumably does the permutation test on the current model and data.

Explain Output Structure: As seen in `explain.json` (presumably for 10m model) and `explain_30m.json`, the format is:

```json
{
  "ts": "2025-08-31T23:45:04+00:00",
  "top_contributors": [
    "dev (+0.3661±0.0130 AP)",
    "anom_fused (+0.0070±0.0177 AP)",
    ...
  ],
  "all_features": [
    { "feature": "dev", "mean": 0.36614, "std": 0.01300 },
    { "feature": "anom_fused", "mean": 0.006987, "std": 0.01769 },
    { "feature": "dev_roll_std", "mean": 0.0, "std": 0.0 },
    ...
    { "feature": "event_count_24h", "mean": 0.0, "std": 0.0 }
  ],
```

```
  "n": 3
}
```

Here, `"n": 3` might indicate they evaluated on 3 data points? That would be a very small sample – possibly they only explained on the 3 most recent events or a small holdout. This could also be because at that time only 3 events were available to evaluate AP on. If you calculate AP on only 3 points, the resolution is coarse, which might explain why only one feature stands out. This is a bit speculative, but the `n` is likely number of positive events or some size of test set.

Example Interpretation: From the above, we interpret that *"The price deviation (dev) was by far the most important feature, accounting for about +0.366 AP (±0.013) in the 10m model's accuracy. In contrast, the anomaly ensemble score contributed very little (+0.007 AP), and other features had negligible or zero impact on this particular evaluation."* This suggests that in the current state of the model, it's essentially relying almost solely on `dev` to make predictions – an insight that might prompt the researcher to either incorporate more varied training data (so that other features become useful) or be cautious that the model might be too narrowly focused. It's possible that at the time of explanation, most events in the data were simple depegs where `dev` alone was a strong signal. If there were scenarios where `dev` was small but something like `oracle_ratio` was off (maybe indicating an oracle delay), then `oracle_ratio` would matter more. The explainability output can highlight such cases if run at different times.

SHAP vs Permutation: If they had used SHAP (e.g. TreeSHAP from XGBoost), the output would likely be in terms of contribution to log-odds or probability, not AP. The choice of AP as the metric is interesting – it directly ties importance to the model's end-goal (ranking events). Permutation importance's drawback is it might underestimate importance of correlated features (if `dev` and `anom_fused` were correlated, messing up one might not drop AP if the other still provides the signal). But since `anom_fused` partly uses `dev` internally, they are indeed correlated. The result showing `anom_fused` near zero might be because once `dev` is gone, maybe anom_fused also becomes uninformative or vice versa. But to properly measure, sometimes one would remove groups of features together. Nonetheless, the output gives a clear story.

Using Explainability: The system likely uses these explain outputs in the **incident snapshot and analyst note generation**. In the code for building the snapshot (`build_analyst_note_v2`), it reads `top = json.loads(EXPLAIN_JSON.read_text())` to get the latest explain results[76][77]. Then in the incident report markdown, it includes a section "## Top Contributors" and dumps the JSON of `top`[78]. Indeed, the snapshot will have something like:

````
## Top Contributors
```json
{
  "ts": "...",
  "top_contributors": [
    "dev (+0.3661±0.0130 AP)",
    "anom_fused (+0.0070±0.0177 AP)",
    "dev_roll_std (+0.0000±0.0000 AP)"
  ],
  "all_features": {
     ...
  }
}
````
````

This is a bit raw for a report, but it ensures all data is there for an analyst. Possibly they intended to format it nicer eventually. But the presence of it confirms that the snapshot includes which features were most responsible.

Interpretation wise, if an analyst sees the top contributors for a given event were `dev` and maybe something else in another scenario, they can glean the cause. For example, if a future incident had a large liquidity drop but small price move, perhaps `tvl_outflow_rate` might show some importance. That would tell us the event was triggered by liquidity exit rather than price slip.

Explainability for Transparency: Given this is a research-grade system, the inclusion of explainability is to allow model validation and to avoid black-box predictions. If the model ever flags something as high risk, the team can check the explain output to ensure it aligns with domain intuition (e.g. it should be because dev was high or something obviously alarming, not due to a random fluctuation in an unrelated feature). If it's not aligning (like model focuses on a weird feature), that could indicate a bug or overfitting.

Technical Implementation: The code likely uses scikit-learn's `permutation_importance` or a custom routine. The snippet calls `explain_forecast_30m(n_repeats=8)`, which likely: - loads the model and a recent batch of data (maybe last 100 points or some tail), - calculates baseline AP, - for each feature, shuffles it N times and calculates AP each time, - computes mean and std drop (or difference), - returns a dict of feature contributions (maybe sorted and truncated for top_contributors vs all_features list).

The `EXPLAIN_JSON` file path and `EXPLAIN_30M_JSON` are likely updated regularly (maybe every run or at least in nightly report). In [25], after the nightly report, they call `explain_forecast_30m` and print the `top_contributors` to console[79][75], and presumably write it to file.

Integrating with Dashboard: The dashboard could in principle display feature importance visually (like a bar chart of contributions). Not sure if they implemented it, but they did make the data available. In the snapshot .md file, they include the JSON as text, which is at least accessible.

In summary, the explainability module provides **post-hoc feature attribution** for the risk predictions. It identifies which input features were most influential in the model's decision, measured by impact on prediction performance. The current evidence suggests `dev` (price deviation) is usually dominant – which makes sense given the nature of depeg events – but the framework is in place to catch other drivers too. As the system evolves (or if applied to different contexts), this could highlight, for instance, that for some stablecoin, liquidity outflow is a leading indicator of trouble, etc. This information is fed into the incident reports to aid analysts in understanding model rationale and also backtesting the model's logic against real events.

8. Meta-Policy Layer (MCP) and Automated Actions

The **Meta-Policy Layer** (sometimes referred to as MCP - Meta Controller Policy) is the rule-based decision engine that sits on top of the machine learning components. Its role is to interpret the outputs (anomaly scores, risk forecasts, severity assessments, data health checks) and decide on **actions or states** for the system. It encompasses the policy endpoints `/policy/decide`, `/policy/retrain_check`, and `/policy/snapshot`, as well as internal routines for alert triggering and model maintenance.

Let's break down each piece of this layer:

`/policy/decide` – Alert Decision Making

This is a **POST endpoint** where the client can send the current context (e.g., whether data feeds are fresh and recent forecast values) and receive a decision on system alert status[23][24]. Essentially, it

implements the logic: *Given the current risk levels and data quality, should the system raise an alert, and if so, of what severity?*

Inputs: The example in the documentation shows a JSON payload: `{"feeds_fresh": true, "recent_forecasts": {"poolA": 0.62}}`[80][81]. This suggests the endpoint expects: - `feeds_fresh`: a boolean indicating if upstream data (like price feeds, on-chain queries) are up-to-date. If this is false, the system might decide it's not confident to make any judgment. - `recent_forecasts`: an object mapping pool identifiers to their recent risk probability (calibrated). Possibly the highest risk in the last short period or the current risk. Here `poolA` had 0.62 (62%) predicted risk.

Decision Logic: Based on such input, the meta-policy likely applies thresholds to determine an **alert level**: - If feeds are not fresh (`false`), the response might be "inactive" or a decision to wait (no action) because stale data could cause false alarms. - If feeds are fresh and at least one pool has a high risk probability, it triggers an alert. The system might have defined risk level cut-offs for different alert levels: - e.g., Probability > 0.8 = **Red Alert** (imminent critical event) - Probability > 0.5 = **Orange Alert** (elevated risk) - Probability > 0.2 = **Yellow** (noticeable but low probability) - Else = Green/normal.

From the example, 0.62 might cross a threshold for “orange” (just guessing: maybe >0.6 is orange). The endpoint likely returns a structure describing the decision. It could be as simple as:

```
{"poolA": {"level": "orange", "action": "monitor"}}
```

or perhaps a consolidated decision like `{"level": "orange", "message": "Medium risk detected for poolA"}`. The documentation calls it *meta-controller policy decision*, implying it might not just be one pool but an overall system state. If multiple pools have issues, it might pick the highest.

The code (`decide_latest`) likely aggregates the last N minutes of data for each pool and assigns each an alert level. The Nightly report uses `decide_latest` to list incidents (it filtered `reds` and `oranges`)[25][26]. We can infer that: - `decide_latest(n_tail=12)` for example might look at the last 12 rows (maybe ~1 hour if data is every 5 minutes) and determine if any incidents (in terms of risk crossing thresholds or anomalies) occurred. - It returns a DataFrame with columns like `ts, pool, level, anomaly metrics, forecast probabilities` for those incidents. - They consider the highest level in that DataFrame as the current alert. If none are orange/red, presumably the level is “green” (no alert).

The meta-policy might also incorporate **cooldowns**: e.g., if an alert was just raised 5 minutes ago, avoid flapping alerts repeatedly. There's mention in `trigger_alerts_if_needed` of a `cooldown_s: 600` (10 minutes) to avoid duplicate alerts[82][83]. Also mention of an `ack` requirement for red alerts (someone must acknowledge a critical alert)[84][85], which implies the system won’t spam on something that’s awaiting ack.

In essence, `/policy/decide` likely encapsulates the logic: *if risk forecasts indicate a problem and data is reliable, classify the situation as green/yellow/orange/red*. If orange/red, further actions (like snapshot, sending to contract) are triggered.

`/policy/retrain_check` – Model Retraining Signal

This **GET endpoint** returns whether models should be retrained now. It uses: - The output of `compute_feature_drift()` (which we detailed in Section 2): giving `drift: true/false`, reason, metrics. - Possibly also time-based schedule (nightly).

The logic as per code:

```
drift = compute_feature_drift()
if drift.get("drift", False):
    return {"should_retrain": True, "reason": "feature drift", "drift": drift}
return {"should_retrain": True, "reason": "scheduled nightly or drift threshold (PR-AUC drop) hit", "drift": drift}
```【24†L253-L262】【24†L259-L267】.
This is interesting: it always returns `should_retrain: True` even if drift is false, just giving a different reason. This suggests they have set it to retrain at least once per day no matter what (the "scheduled nightly" reason). In a more refined system, one might return false if no drift and not the scheduled time, but here perhaps they simplified or always want to retrain daily. The included drift info can help the caller decide if it's an urgent retrain or just routine.

So, a client (or an internal scheduler) hitting `/policy/retrain_check` will get a JSON:
```json
{
  "should_retrain": true,
  "reason": "feature drift",
  "drift": {
     "drift": true,
     "reason": "feature drift",
     "thresholds": {"ks":0.2,"psi":0.25},
     "metrics": { "dev": {"ks":0.25,"psi":4.92}, ... },
     "n_train": ..., "n_recent": ...
  }
}
```【41†L109-L117】【41†L109-L119】.
If drift was false, it would still give `should_retrain: true` but reason "scheduled nightly or drift threshold (PR-AUC drop) hit", and drift metrics included indicating no drift【8†L264-L272】【8†L266-L274】.

The meta-policy uses this to kick off retraining workflows. For example, the code might have a loop that every hour or so checks this endpoint and if `should_retrain` is true, it triggers the pipeline to retrain models (and then likely resets or notes last retrain time to not do it again immediately).

`/policy/snapshot` – Incident Snapshot and Reporting
This **GET endpoint** provides a "one-page JSON + rendered note" of the latest incident snapshot【8†L266-L274】【8†L271-L279】. Essentially, whenever a significant incident occurs, the system compiles a report (snapshot) containing:
- A detailed **analyst note** (which may be autogenerated or manually added context).
- Top features contributing (the explainability JSON).
- Network context (if applicable).
- Citations (if any external data or references were used).
- Some state metadata (like whether ack is required, etc.).

The code that builds this is `write_incident_snapshot(level, note, citations, extras)`【30†L45-L53】【30†L46-L54】. It gathers:
- `note`: which might be an analyst's free-form notes or automatically generated text summarizing the event (perhaps using a template: e.g. "Price dropped X% in Y minutes due to ...").
- `citations`: maybe references to external news or on-chain events that correlate (if integrated with a knowledge base).
- `net`: network features (they call `compute_network_features()` to see if other pools

are related)【30†L78-L86】【30†L84-L92】. They produce a `net` dict of neighbor_max_dev, neighbor_avg_anom, best lead-lag correlation etc. This indicates if the event might have been caused by or caused anomalies in other pools (like if two pools are highly correlated, one may lead the other).
- `top`: explanation features from `EXPLAIN_JSON`【30†L88-L96】.
- `extras`: state info like `{"hash": h, "requires_ack": True}` (for a red alert requiring human acknowledgment)【30†L47-L55】.

It then formats a markdown with sections:
  - **Incident Snapshot – LEVEL** (with a timestamp)【30†L93-L101】.
  - **Analyst Note** (if any text provided)【30†L97-L100】.
  - **Top Contributors** (the JSON of top features)【30†L99-L106】.
  - **Network Cue** (the JSON of network context)【30†L100-L104】.
  - **Citations** (list of references, if any)【30†L103-L107】.
  - **State** (the extras JSON with ack hash etc)【30†L105-L113】.

This is then written to a markdown file `incident_YYYYMMDDTHHMMSS.md` in the output directory【30†L113-L118】, and likely also stored in `events.json` as part of the incidents list.

The `/policy/snapshot` endpoint likely reads the latest such snapshot file and returns its content (or a link to it). It's described as returning "one-page JSON + rendered note" – possibly meaning it returns a JSON with both the raw data and maybe a pre-rendered HTML or text of the note.

So when a user (like an ops team member) calls `/policy/snapshot`, they get the incident report of the most recent high-level alert, which includes all the details the system gathered. This is extremely useful for quickly diagnosing what happened without having to manually gather all information.

Alert Lifecycle and Retraining triggers
Alert Escalation: The meta-policy distinguishes levels:
- **Red alerts**: The code sets `requires_ack = True` and uses a branch for them【30†L37-L45】【30†L49-L57】. This means a red alert is considered critical; the system will keep the alert active and likely not send another red for the same issue until someone acknowledges it (perhaps via an `/ack/{hash}` URL they provided in the post data to webhook【30†L39-L47】【30†L51-L59】).
- **Orange alerts**: are significant but not requiring ack. The system will snapshot them but perhaps auto-clear after some time.
- **Green (no alert)**: the normal state, no action except perhaps routine logging.

Integration with Smart Contract: The meta-policy likely decides when to call the smart contract. It could either:
  - Automatically call `submit_data_block` for every orange/red incident (especially red).
  - Or it might call only for red (the highest severity).

Given the example contract call we saw had severity=3 (which might correspond to a borderline event), it suggests even severity 3 events were submitted on-chain. Possibly their policy is to submit any event of severity ≥3. That corresponds to events they consider "real incidents" (since they calibrated severity such that 3 is meaningful). Orange might correspond to severity 3 and red to 4-5. If so, then yes, all severity ≥3 go to chain.

The code doesn’t explicitly show where they call the contract, but one could embed it in the alerts pipeline: after snapshotting, if level is red (or >= orange), call contract. The details needed (block_hash, anomaly_bps, novelty_bps, severity, recipient) are all available at that point:
- `block_hash`: they can compute from the payload (as shown in the contract test snippet)【3†L509-L517】【3†L519-L527】.
- `anomaly_bps` and `novelty_bps`: presumably anomaly BPS = `anom_fused * 10000` (capped 10000) and novelty BPS similarly maybe from one of the detectors that measures outlierness novelty. They likely combine or choose one of the detectors as "novelty" – maybe LOF or OCSVM could serve as novelty measure, or even neighbor_avg_anom from network features could reflect novelty relative to others. The contract expects separate anomaly and novelty values【24†L135-L143】. Possibly they define:
   - anomaly_bps = absolute price deviation in basis points (i.e. if price at $0.98, anomaly=200 bps),
   - novelty_bps = perhaps the difference in anomaly vs correlated pools, or an anomaly score relative to historical distribution (maybe something like if none of the known patterns matched, novelty= high). Not entirely sure, but since they had novelty 430 in example, maybe novelty was 4.3% difference between pool price and oracle or something unusual.
- `severity`: the severity 1–5 from the severity model.
- `recipient`: likely their own address or a pre-configured recipient (maybe the team’s multisig or the data provider’s address). In the example, they used the same address as treasury so it minted to itself【4†L543-L550】【4†L551-L559】.

Drift Handling & Retraining: The meta-policy also covers model maintenance:
- It runs drift checks on features as described. If drift is true, it triggers retrain.
- It also appears to always retrain nightly. The text "scheduled nightly or drift threshold (PR-AUC drop) hit" suggests they intended to also monitor model performance (like if PR-AUC of detectors or forecasters drops below some threshold) as a trigger for retraining, although the current code doesn’t explicitly compute PR-AUC drop beyond nightly winner detector (maybe they could compare winner’s PR-AUC vs previous day).
- The actual retraining process is likely kicked off by a separate scheduler or by the main loop if it sees `should_retrain: true`. Possibly a function `train_if_needed(force=False, min_hours=4.0, min_new_rows=400, label_drift_thr=0.15)` is mentioned in logs【30†L159-L167】. It likely uses conditions like if at least 4 hours passed and 400 new data rows or some label drift threshold, then retrain. The meta-policy’s retrain_check feeds into such logic.

- When retraining triggers, the system will:
   * Sample new data (the live CSV has grown),
   * Recompute labels (with `_ensure_labels`),
   * Retrain forecaster models,
   * Retrain/calibrate severity maybe if there are new events,
   * Possibly refit unsupervised detectors if needed (though things like IF could be refit occasionally, not sure if they do),
   * Save new model artifacts,
   * Update RUN_META.json (with new version info, timestamp of retrain, etc. – they do log `write_run_meta()` after retraining or certain steps【40†L68-L76】【40†L69-L77】),
   * And perhaps clear drift (basically now what was “recent” becomes part of “train”, so drift metrics would reset until distribution shifts again).

Policy in Code vs API: The endpoints allow external components to query decisions, but much of this likely runs internally:

- The **dashboard** might call `/policy/decide` periodically to know if it should show a big alert banner.
- Automated scripts might call `/policy/retrain_check` on a schedule to see if retrain needed (though since it's always true with current logic, they might just do daily).
- The system itself, since integrated, could just call internal functions rather than via HTTP.

Network Effects: The meta-policy also tries to catch *broader context* via network features. The function `compute_network_features(win=60, max_lag=10)` builds features per pool for cross-pool analysis【34†L64-L72】【34†L74-L82】:
   - It maintains a rolling window of dev and anomaly for each pool,
   - For each pool, finds the maximum absolute dev of any *other* pool in that window (`neighbor_max_dev`),
   - Averages anomaly of others (`neighbor_avg_anom`),
   - Computes the best cross-correlation and lag between this pool's dev series and any other pool's dev (`corr_best`, `lead_lag_best`)【34†L93-L101】【34†L103-L111】.
   - These indicate if the pool's movement might have been *led by* or *lagging* another pool. For example, if DAI/USDC and USDC/USDT are highly correlated, and one moves first by 5 minutes, they capture that.
   - They attach the latest values of these to each pool's record.

This can inform policy: if an anomaly in pool A was simply caused by pool B's depeg (which happened first and is bigger), one might mark pool A's incident as secondary. Or if a pool's anomaly had no correlation (novelty), that might make it more severe as mentioned.

Snapshot Inclusion: The snapshot adds a "Network Cue" showing one row (the worst neighbor stats for the pool of interest)【30†L78-L86】【30†L88-L96】. In an example output (not provided but we can imagine):
```json
{
  "pool": "USDC/USDT_univ3",
  "neighbor_max_dev": 0.0012,
  "neighbor_avg_anom": 0.05,
  "lead_lag_best": -3,
  "corr_best": 0.9,
  "win": 60,
  "max_lag": 10
}
```
This could mean: some other pool had at most 0.12% dev, average anomaly 5%, and the best correlation is 0.9 at lag -3 (maybe meaning this pool lags another by 3 minutes with correlation 0.9). That would indicate the event might be a follow-on effect.

Meta-Policy and Data Integrity: Another aspect is ensuring actions happen only when data is trustworthy. The `feeds_fresh` flag shows they incorporate data staleness. They might also check if the ML models have been loaded and are ready (the `/ops/readyz` endpoint possibly monitors that)[88][89]. If not ready, policy might refrain from decisions. Additionally, if the forecast data itself hasn't updated recently (maybe due to pipeline stuck), they wouldn't raise an alert because it could be a false stagnation.

Logging and Audit: The meta-policy outputs are logged (e.g. printing no orange/red alerts, posted X alerts to webhook, etc.)[26][90]. It also keeps an `_LAST_ALERT` state with hash and time to avoid duplicates[82][83]. If an identical alert (same content hash) happens and it's within cooldown, it

suppresses it[83]. For red alerts, requires_ack stays false until ack, meaning if an ack endpoint is implemented, after ack they might allow new red alerts. If not acked, they might not spam but will keep posting or keep it flagged.

Summary: The Meta-Policy Layer is the **brain** that translates model outputs into operational decisions: - **Decide**: Is the system all clear or is there an incident? If incident, how severe? It uses thresholds and logical rules on calibrated risk and severity. - **Snapshot**: If incident, compile all relevant info into a report for users (with context and explanation). - **Alerts**: Manage sending alerts out (e.g. via webhook, which could be Slack/Discord notifications). We saw code posting JSON to a `webhook_url` with the alert payload[90]. - **Acknowledge & Cooldown**: Manage alert lifecycle to avoid spam and require human intervention for critical ones. - **Retrain decisions**: Monitor drift and schedule to keep models fresh, and trigger retraining when needed.

It's essentially a rule-based expert system on top of the ML system to ensure it behaves in a stable, interpretable way. This separation of concerns is good: the ML does scoring, the meta-policy decides actions – so one can adjust thresholds or policies without retraining ML, and vice versa.

9. Dashboard and User Interface

The Depeg Sentinel comes with a **dashboard** (a Streamlit web application) that allows users (researchers, analysts, or operators) to monitor the system in real-time or replay past data. This dashboard visualizes the key outputs – anomaly scores, risk levels, and incidents – and provides controls for the user to configure the view.

Components and Files: The main files supporting the dashboard include: - `dashboard.py` – the Streamlit app script[91][92]. - The **artifacts** directory with output files that the dashboard reads: - `live_dataset.csv` – continually updated dataset of recent data points (with timestamp, pool, features like dev, and anomalies). - `forecast_10m.parquet` (and similarly `forecast_30m.parquet` if separate) – contains model predictions over time for each pool. - `events.json` – log of incidents/events that have occurred. - Calibration files (`calibration_10m.json`, etc.) and performance artifacts (e.g., detector_pr_auc.json, images) – used for offline analysis or downloadable from the UI. - `RUN_META.json` – contains meta info (versions, config, counts) which the UI might display to know the environment.

The dashboard is designed to operate in two modes: - **Live Mode:** Connected to an actively running backend. In this mode, the CSV and other artifacts are being updated by the running sentinel process. The UI will periodically refresh and show the latest state. - **Demo Mode:** If no live data is present, the dashboard can generate or use synthetic data to demonstrate how the system works. The code snippet shows a toggle for "Demo Mode"[93][94], which defaults to True if `LIVE_CSV` does not exist (meaning there's no real data file)[93]. In demo mode, the code uses pre-baked or generated data: - There's mention of `monkeypatch` and `_mk_live` in tests generating synthetic trends[95][96], likely the demo uses a similar approach. - Essentially, demo mode might simulate a scenario (like a gentle drift into an anomaly) across two example pools "USDC/USDT-uni" and "DAI/USDC-curve"[97].

Layout and Controls: The dashboard's layout has: - A sidebar with controls: - **Demo Mode toggle:** Switch between using real artifact files vs. synthetic data[93]. - **Timezone selection:** They allow the user to view timestamps in various timezones (UTC, Denver, etc)[98]. This is helpful for localizing times. - **Lookback Window (hours):** A slider to choose how many hours of history to display on charts (1 to 168 hours, default 24)[99]. This restricts the data view to recent N hours. - **Deviation threshold |abs(dev)|:** A number input to set the threshold line for price deviation (like 0.005 by default, meaning 0.5%). Possibly this draws a reference line on charts or is used to filter incidents (like show all points beyond this as red). It's user-adjustable[99]. - **Fused anomaly threshold:** Another input (default 0.90) to mark where fused

anomaly is considered triggered[100]. This might again be used in highlighting. - **Auto-refresh seconds:** A slider (0 to 120 sec, default 30 sec) controlling the interval for auto-refresh of data. If set to 0, auto-refresh is disabled (useful for static analysis)[36][101]. - The sidebar also shows the **Paths** of the data files currently in use (for transparency, they print the output dir path, and names of live CSV and forecast file)[102][103].

- Main view:
- **Metrics at top:** They have five columns displaying key metrics[104][105]. If data is available:
    - *Last update:* timestamp of last data point (converted to local timezone choice)[106].
    - *Data Status:* if live is empty, show "No live data", else maybe "OK" or blank (the code sets it to "—" if no data)[107][108].
    - *Fused anomaly (avg now):* The average `anom_fused` value of the selected pools at the latest timestamp[109][110].
    - *10m risk (avg):* The average 10m calibrated risk across selected pools at latest time[52].
    - *30m risk (avg):* Similarly for 30m risk[52].
    - *Incidents (total):* Number of incidents in events log (they count `len(events.get("incidents", []))`)[111].
- These metrics give a quick summary of system state. For instance, if fused anomaly avg is 0.8, that's high; if risk avg is, say, 50%, that's concerning; incidents total tells how many events occurred (to know if something triggered recently).

- **Pool selection:** A multi-select widget allows choosing which pools to display (the default selects up to 3 pools if more available)[112][113]. This filters the data shown in charts and affects the average calculations above.

- **Charts and visualizations:** Although not fully shown in snippet, typically a Streamlit dashboard for this scenario would include:

    - A time-series chart of deviation over time, perhaps with threshold lines drawn (the `dev_thr` from sidebar).
    - A parallel time-series of `anom_fused` over time, perhaps with threshold line (0.9 if chosen).
    - Possibly subplots of the individual detectors' scores vs time (to see which detector triggered).
    - Risk forecast plots: the predicted 10m and 30m risk probabilities over time for each pool, maybe as separate lines or area under curve.
    - If multiple pools selected, they may overlay them or show separate charts per pool (the code seems to group by pool and then tail the timeframe, etc.).
    - Perhaps a table or list of incidents (with time, pool, severity). Since they compute `events` from `events.json`, they may show a table of incidents or highlight them on charts (like vertical markers).
    - The uploaded `events.json` might have each incident's summary and severity; the dashboard could display those in a section listing e.g. "Time X: USDC/USDT - severity 3 - summary...".

  The code indicates they transform `events = load_json(EVENTS_JSON)` and then `n_incidents = len(events.get("incidents", []))` for metric[111]. They likely also use `events` to draw markers on graphs or populate a section (though snippet not fully visible).

- **Auto-refresh behavior:** If refresh_sec > 0 and `st_autorefresh` is available, they call `st_autorefresh(interval=refresh_sec*1000, key="sentinel_refresh")`[114][115]. This causes the page to auto-update on that interval. The metrics and charts will update to reflect new data appended to live CSV or events JSON by the backend. The user can set refresh to 0 to pause updates (for example, to investigate something without the view shifting).

- **Data reading and caching:** They define loader functions `load_live`, `load_forecast`, `load_json` which likely use caching so as not to constantly reread large files. Actually we see `@st.cache_data(ttl=20)` on `list_artifacts`[116], meaning they cache listing images and zips for 20 seconds. Possibly they did similar for load_live etc (not visible in snippet but likely cached with a short TTL to not overload disk).

- After loading data, they do grouping by pool and filtering by selected pools for display[117][109]. They compute `last_ts` and its local time for display, as we saw.

- They also prepare numbers for fused anomaly now, p10 now, p30 now by averaging across selected pools at last timestamp[109][52]. These appear in the metric columns.

- Then they show these metrics using `_metric` (possibly a helper function to format metric nicely), as seen with `_metric("Fused anomaly (avg now)", f"{fused_now:.3f}")` etc[118]. This likely uses Streamlit's metric or write to show nicely.

- Following metrics, they likely draw the main plots. The code for plotting likely uses Plotly (they import plotly.express as px and graph_objects as go)[119]. They might create interactive charts for each pool's timeseries. For example, a combined chart with dev on one y-axis and risk on another, or separate charts:

    - Price deviation vs time (with threshold).
    - Anomaly scores vs time (maybe stacked or separate).
    - Risk probability vs time (with incidents marked).
    - Possibly distribution plots or calibration curves in a separate page/tab if needed (though not mentioned, they may not expose that in UI).

The `list_artifacts` function finds any PNG images in the artifacts directory and any zip files, which might be offered as downloads (like calibration plots PNG, or a zip of logs)[120]. They might allow the user to download the nightly report PDF or other artifact from the UI.

Modes and Examples: - In **live mode**, one would run the Sentinel backend (which populates live_dataset.csv continuously) and then run `streamlit run dashboard.py`. The UI connects by reading the files the backend writes. The auto-refresh ensures near-real-time update (with maybe a slight lag depending on refresh interval). - In **demo mode**, if someone just wants to try the app without a running backend, they launch `streamlit run dashboard.py` and since no `live_dataset.csv` exists, it toggles demo. The code likely generates a synthetic CSV with some patterns (as in test fixture `_mk_live` created a sine wave or linear dev and anom_fused gradually increasing to simulate a depeg)[97]. Demo mode might also disable some features (like obviously it won't call actual smart contract or have no real incidents). - The user can interact: choose different pools to focus on, adjust thresholds (maybe to see how events detection changes, though thresholds in UI might just be visual markers; they might not retroactively change events but could highlight points beyond threshold). - The user can also scroll back in time by adjusting lookback (e.g., last 72 hours). - If an incident happens while the user watches, the incident count metric will increment and potentially a marker or highlight appears on the chart at that

time with severity label. The UI might also push a notification sound or visual if a red alert triggers (not sure if implemented, but possibly could highlight the header in red or something).

Refreshing Artifacts: Some artifacts like calibration_*.png or detector_pr_auc.png are produced in nightly report. The function `list_artifacts` collects them. They might show these images in a separate section or allow user to open them. Possibly, below the main charts, they might have an "Artifacts" section listing links or images. For instance, "Latest Detector Performance: see detector_pr_auc.png" and "Calibration plots for 10m and 30m".

Local vs Cloud Deployment: Streamlit can run locally or be deployed to Streamlit Cloud. The question mentions launching local or Streamlit Cloud – likely instructing how a developer can run it. E.g.,: - Locally: `streamlit run dashboard.py` will start a server on `localhost:8501`. - On Streamlit Cloud: one could upload the repo and it will run automatically, but one might need to ensure demo mode or link it to a running backend. More likely, Streamlit Cloud usage would be demo mode unless it's also connected to a persistent backend.

Cache & Interval: They use `@st.cache_data(ttl=20)` which caches outputs for 20 seconds[116]. For heavy file reading (like reading a possibly large CSV), caching avoids doing it too often. With auto-refresh 30 sec, caching 20 sec ensures at most one re-read per refresh or two. They might similarly cache `load_live` so it doesn't re-read CSV on every widget interaction.

Tech Stack: It's pure Python with Streamlit. Dependencies are those needed for charts (Plotly) and data (pandas, numpy).

Dashboard Summary Example: When running, the dashboard might look like: - Title (maybe none given in code, but could add). - Sidebar to control settings as above. - Top row of metrics: - Last update: e.g. "2025-09-01 15:50:00 UTC", Data Status: "OK", Fused anomaly avg: 0.002 (basically fine), 10m risk avg: 0.1%, 30m risk avg: 0.3%, Incidents total: 2. - Multi-select with pool list [USDC/USDT_univ3, DAI/USDC_univ3, 3pool_curve]. - Charts: perhaps one combined chart per pool: - time series of price (should hover around 1, showing slight dips), - underneath it anomaly score maybe as area or line (with a red horizontal line at 0.90 threshold), - risk predictions maybe as another subplot or overlay as colored region indicating probability (or separate small chart). - If an incident occurred (say at 14:30, USDC/USDT dropped to 0.97), the chart might mark that point with a red dot or vertical line. The events list might show "14:30 UTC | USDC/USDT | fused=1.00 | p10=0.95" meaning at that timestamp fused anomaly was 1 and 10m risk 95% (and presumably an incident). - Possibly a table: "Recent Incidents: [Time, Pool, Severity]" with a short summary or a link to snapshot.

However, in code snippet for Nightly report, they had a section "## Incidents (last window)" listing incidents in markdown[121][122]. The UI might not replicate that but it does have `n_incidents` count and presumably one could expand events in the UI.

Artifact Directory & Refresh: The default artifact dir might be `./artifacts` (based on config in environment, we saw `ART` path in logs[123][124]). The Streamlit app likely expects environment variable `OUT_DIR` or similar to point to artifacts. In code, `OUT_DIR = Path(os.getenv("OUT", "./outputs"))` or similar might be set. They then do `out, live = _mk_live(tmp)` in tests to simulate that. In real run, perhaps environment variable `OUT` is set to the artifacts directory.

In `dashboard.py`, they refer to `OUT_DIR`, `LIVE_CSV`, `FORECAST_PQ`, etc. Possibly they define those at top, e.g.:

```
OUT_DIR = Path(os.getenv("OUT", "./artifacts"))
LIVE_CSV = OUT_DIR / "live_dataset.csv"
```

```
FORECAST_PQ = OUT_DIR / "forecast_10m.parquet"
EVENTS_JSON = OUT_DIR / "events.json"
ARTIFACTS_DIR = OUT_DIR  # or a subdir for images
```

The UI prints `st.text(f"{OUT_DIR}")` to show where it's reading from, and `st.text(f"{LIVE_CSV.name} | {FORECAST_PQ.name}")` to show which filenames are active[102][103]. This is to avoid confusion which data is being used (especially if user set OUT env).

Example Launch: - **Local:** Ensure you have Python 3.10+, install requirements (pandas, numpy, streamlit, plotly, scikit-learn, xgboost, web3, etc – see Section 13). Start the backend sentinel (if you want live mode) which writes artifacts. Then in another terminal, run `streamlit run dashboard.py`. A browser opens at `localhost:8501` showing the UI. If you just run the UI without backend, toggle "Demo Mode" to see sample data. - **Streamlit Cloud:** Push the repository to a platform, set environment variables as needed (like `MOCK_MODE=1` for demo maybe, or provide a sample artifacts folder zipped). The app will run – likely in demo mode since no artifacts persistent – showing synthetic data demonstration. If wanting to connect to live data on cloud, one would need the sentinel to also run on some server accessible by the Streamlit app (or hack by writing to cloud storage). But that's beyond scope; presumably, for demonstration, you'd just run demo mode or run both on the same host.

Caching and Intervals: The TTL=20 on cache means the UI will re-read files at most every 20 seconds from disk. If auto-refresh is 30 sec, that's fine. If auto-refresh is set to 5 sec by user, the TTL might cause it not to fetch new data until at least 20 sec passes. That's probably okay for anomaly detection timescales.

Dashboard Dependencies: The UI requires: - Python libs: streamlit, pandas, numpy, plotly, altair (if used), maybe streamlit_autorefresh (they import it)[92], statsmodels if any smoothing needed (they imported but maybe not used)[125]. - If interacting with web3 or backend, not directly; UI doesn't call contract.

In summary, the dashboard provides: - **Live visualization:** to observe stablecoin pools, anomalies, and risk in real time, with auto-refresh. - **User controls:** to adjust time window, thresholds, and toggle between real and mock data. - **Incident logs:** so the user can see if any incidents were detected and dig into details (with possible link to snapshot or at least count and severity). - **Demo capability:** so the system can be demonstrated without needing to run on actual DeFi data.

It's an essential tool for both demonstrating the system (to stakeholders) and for researchers to monitor how the system is performing, view when it triggers, and perhaps identify false alarms or missed events visually and adjust accordingly.

10. Smart Contract Integration (Tokenized Rewards)

One novel aspect of Depeg Sentinel is its integration with a **smart contract** on Ethereum (or a testnet like Sepolia, as logs indicate) to record detected incidents and issue token rewards. The smart contract used is called **SentinelDataToken**, and it's written in Vyper. This contract essentially tokenizes the act of reporting an anomaly – minting a certain amount of **SDT tokens** whenever a data block (an incident report) is submitted, thereby incentivizing the detection and reporting of depeg events.

Contract Overview: SentinelDataToken is likely an **ERC-20 compatible token** with some custom functions: - It has standard ERC-20 functions: `name()`, `symbol()`, `decimals()`, `balanceOf()`, `transfer()`, etc. (We saw `name=SentinelData`, `symbol=SDT`, `decimals=18` in the deployment output[126][127]). - It maintains an `owner` and a `minter`. At deployment, they set both to the deployer's address (which is the backend system)[126][128]. - It has a `treasury` address and a `treasury_bps` (basis points for treasury

fee, set to 500 i.e. 5%)[126][127]. - The core custom function is `submit_data_block(bytes32 block_hash, uint256 anomaly_bps, uint256 novelty_bps, uint256 severity, address recipient) -> uint256`[129][130]. This is called by the backend when a depeg incident is detected.

Permission and Security: Only the authorized `minter` can call `submit_data_block`. The contract code snippet shows:

```
@external
def submit_data_block(...):
    self._only_minter()
    assert not self.paused
    assert recipient != empty(address)
    assert not self.used_block[block_hash], "dup"
    self.used_block[block_hash] = True
    reward = self._compute_reward(anomaly_bps, novelty_bps, severity)
    t_fee = reward * self.treasury_bps / MAX_BPS
    net = reward - t_fee
    if t_fee > 0:
        self._mint(self.treasury, t_fee)
    self._mint(recipient, net)
    log DataBlockSubmitted(block_hash, anomaly_bps, novelty_bps, severity, reward,
recipient)
    return reward
```【24†L145-L154】【24†L155-L164】.

Key points:
- It checks `msg.sender` is minter (so only the backend, which holds the private key or runs on a node with that account unlocked, can call it)【24†L149-L157】.
- It ensures the contract is not paused (there's a pause mechanism likely for emergencies)【24†L149-L157】.
- It requires `recipient` is a valid address (not 0)【24†L149-L157】.
- It prevents double submission of the same data block by keeping `self.used_block` mapping of block_hash to bool【24†L151-L159】. If the same `block_hash` is submitted again, it will `assert` fail with "dup".
- It then computes a `reward` by `_compute_reward(anomaly_bps, novelty_bps, severity)`【24†L155-L163】, and splits it into `t_fee` (treasury fee portion) and `net` (net to recipient).
- It mints the fee portion to the treasury address and the remainder to the recipient【24†L159-L166】.
- It logs a `DataBlockSubmitted` event with details and returns the total `reward` minted.

Reward Emission Logic: The reward formula as defined in `_compute_reward`:
```vyper
@internal
def _compute_reward(anomaly_bps: uint256, novelty_bps: uint256, severity: uint256) ->
uint256:
    assert anomaly_bps <= MAX_BPS
    assert novelty_bps <= MAX_BPS
    assert 1 <= severity <= 5
    base: uint256 = 10**18  # 1 token unit (assuming 18 decimals)
    return base * (anomaly_bps + novelty_bps) * severity / (MAX_BPS * 5)
```【24†L135-L143】.

Breaking this down:
- `MAX_BPS` is likely 10000 (100.00%). They ensure anomaly_bps and novelty_bps are at most 10000 (so 100% expressed in basis points)【24†L137-L140】.
- severity is clamped 1 to 5【24†L139-L140】.
- `base = 10**18` sets 1 token in smallest units (because 18 decimals).
- Reward = `base * (anomaly_bps + novelty_bps) * severity / (10000 * 5)`【24†L141-L143】.

Simplify: $\text{reward tokens} = \frac{(anomaly\_bps + novelty\_bps) \times severity}{5 \times 10000}$ (since base/1e18 just sets it in token units).
In decimal:
- anomaly_bps + novelty_bps gives a number up to 20000 in worst case.
- Divide by 10000 yields (anomaly% + novelty%) as a decimal up to 2.
- Multiply by severity/5 yields a factor from severity normalized to [0.2,1] (1 gives 0.2, 5 gives 1).
- Multiply those: effectively reward = (anomaly% + novelty%) * (severity/5) tokens.

Example: anomaly_bps=820, novelty_bps=430, severity=3:
(anomaly+novelty)=1250 → /10000 =0.1250, severity/5 = 3/5 =0.6, product =0.0750 tokens.
Indeed in the test output, reward was 0.075 tokens【4†L545-L553】【4†L575-L579】.

So:
- If a pool had 0 anomaly (no price deviation) but novelty is high (like novelty detection triggered something weird with no price move), anomaly_bps=0, novelty_bps maybe some measure of weirdness. The reward would scale with novelty% alone. But likely, novelty in practice might refer to some metric like "how out-of-pattern is this event"; if none, novelty might be low.
- If both anomaly and novelty are max (100% anomaly, completely novel scenario) and severity=5 (worst), then reward = (10000+10000)/10000 * (5/5) = 2 * 1 = 2 tokens. So 2 SDT tokens would be minted for a truly extreme event.
- If a moderate event: say 2% anomaly (200 bps), novelty moderate 100 bps, severity=3, reward = (300/10000)* (3/5) = 0.03*0.6 = 0.018 tokens.

These token amounts are quite small (fractions of a token). The idea might be that the token isn't meant to be high value per event; it’s symbolic or accumulative. If a watchman is continuously detecting issues, they could earn tokens over time. Or if multiple independent agents ran the sentinel and only the first to report gets rewarded (if it was decentralized), these tokens could create a competition to report anomalies fastest. In our case, since the sentinel is automated and itself calling the contract, it’s more like an audit log token.

Treasury Mechanism: `treasury_bps`=500 means 5% of reward goes to a treasury address. In the test, they set treasury = deployer too, so the same address got both pieces[132][86]. In a real deployment, treasury might be a DAO or project fund. For each submission:
- The contract mints 5% of reward to treasury, and 95% to the recipient.
This aligns incentives: a portion goes to project maintainer or to fund future development, while the majority goes to the reporter.

DataBlockSubmitted Event: It logs the event with all input fields plus the reward and recipient[73]. So off-chain observers or a blockchain explorer can see:

DataBlockSubmitted(block_hash = 0xa0ddeab7... (some hash), anomaly_bps = 820, novelty_bps = 430, severity = 3, reward = 75000000000000000 (which is 0.075 *10^18), to = 0x7E5F...Bdf (recipient address))``[73]. This event is critical for transparency – anyone can verify that at a certain

block/time, the system reported an anomaly with those parameters and minted that reward. Theblock_hash` presumably is a hash of the incident data payload (timestamp, pool, values) to uniquely identify it without storing all data on-chain. Off-chain, the actual data might be retrievable or at least can be matched if someone has the same data.

Web3.py Integration: The backend uses Web3.py to interact with the contract. The code snippet from the Contract.pdf shows how: - They compile the Vyper source to get ABI and bytecode[133][134]. - Deploy it (either on a test Ethereum tester or a real network if configured)[135][136]. - After deployment, they call the contract's functions through Web3: - token.functions.name().call() etc to verify deployment[137]. - They prepare a payload dictionary for the event and do blk_hash = w3.keccak(text=json.dumps(payload, sort_keys=True)) to get block_hash[38]. - Set anomaly_bps, novelty_bps, severity, recipient, then call:

```
fn = token.functions.submit_data_block(blk_hash, anomaly_bps, novelty_bps, severity,
recipient)
tx_hash = fn.transact({"from": sender})
```

or if using a remote node with private key, they build and sign transaction (the code handles both unlocked local account vs external via acct)[138][139] and[140][141]. - They wait for rcpt = w3.eth.wait_for_transaction_receipt(tx_hash)[141]. - Then they retrieve balances: balanceOf(recipient), total_supply and see how they changed[142][143]. - They also decode the event with token.events.DataBlockSubmitted().process_receipt(rcpt_submit) to get the event args and print them[144][73].

In a production scenario, this integration implies: - The ML backend must have access to a web3 provider (RPC URL) and the private key of the authorized minter account. In logs, they mention WEB3_RPC_URL and PRIVATE_KEY env variables[145][146]. If PRIVATE_KEY is set, they use it to sign; if not and a local node is used, they expect an unlocked account. - They connect either to a local dev chain (for testing they used EthereumTester or Ganache as fallback)[147], or to a remote chain (like Sepolia per config in RUN_META[148][149]). - Once configured, whenever the sentinel decides to log an event on-chain (likely for severity >= threshold), it will create the payload, compute hash, and call submit_data_block. The returned reward can be logged and the event can be observed on-chain.

Smart Contract Addresses & Vyper Source: The contract's address is printed at deployment (in test it was 0xF2E2...395b)[150]. In a real deployment, one would deploy once and reuse the address in config. The ABI (application binary interface) is provided in SentinelDataToken.abi.json so the web3 interface can be created without recompiling each time.

The Vyper source (not fully shown but partially in [24]) includes standard ERC20 storage like balances, total_supply, plus: - A mapping used_block to prevent duplicates[151]. - Addresses for owner, minter, treasury and a paused boolean likely, plus treasury_bps. - Possibly functions to set minter (owner only), to pause/unpause (owner only), to change treasury or its bps (owner only) – not shown but typical. - The event definition event DataBlockSubmitted(bytes32 block_hash, uint256 anomaly_bps, uint256 novelty_bps, uint256 severity, uint256 reward, address to) which matches what we saw.

Usage and Workflow: When the sentinel triggers a contract call: 1. The system gathers event data (ts, pool, anom and novelty and severity). 2. It JSON-serializes and hashes it to a 32-byte block_hash. This hash is a unique identifier for that data block – ensuring if the same data is submitted twice, the contract will catch duplicate. 3. It calls submit_data_block(hash, anomaly_bps, novelty_bps, severity, recipient_address) via Web3. The recipient_address could be: - The address of the entity to reward. If this system is run by a single operator, they might just reward themselves (like how they used

sender=treasury in test). But conceptually, if different watchers ran the sentinel, the one who found the anomaly could call it with their address to claim reward. - There's potential for decentralization: multiple parties could monitor and race to call the contract first when a depeg happens. The contract will reject second submission of same `block_hash`. But how to ensure consistent `block_hash`? They'd have to hash the standardized payload. This is tricky unless all agree on data format. Given this sentinel is centralized, it's not an open network of reporters yet – but it's a step towards that concept. 4. The transaction is mined; event logged; SDT tokens minted. 5. The backend can optionally listen for the `DataBlockSubmitted` event (via Web3 or a subgraph) to confirm submission and maybe display it on the dashboard (though the dashboard could also just know because it initiated it). 6. The tokens minted can be later claimed or used. The token might have utility in a broader system (e.g., governance or a stake for detection, or just a gamification reward).

Integration Benefits: - **Transparency:** By logging events on-chain, the anomaly detections become publicly verifiable. Anyone can see that at a given block/time, an anomaly was reported with certain severity and scores. This could build trust (the system isn't hiding or modifying events after the fact). - **Immutability:** Once on-chain, the record can serve as an audit trail. If investigating a crisis later, one can check if the sentinel reported it and when. - **Incentives:** The token mechanism could incentivize community participation in monitoring. Currently, the sentinel itself is the reporter, but the architecture could extend to multiple independent bots. The first to report an event gets the reward, encouraging broad coverage and rapid reporting. - **Treasury fund:** The 5% cut could accumulate in a treasury to fund maintenance or pay for false-positive costs etc.

Using the ABI and Vyper Source: The documentation includes the ABI file (which we have). The ABI is a JSON describing the contract's interface (function signatures, inputs/outputs, events). For example, it shows `submit_data_block` expects those 5 parameters[152][153]. A developer can use this ABI with Web3.py by loading it, specifying contract address, and then calling functions as shown.

The Vyper source code is partially shown, demonstrating key logic (we've covered the main logic above). If needed for documentation, one might include a snippet like:

```
# Pseudocode for reward calculation
reward = ((anomaly_bps + novelty_bps) / 10000) * (severity / 5) * 1 token
# e.g., 8.2% anomaly + 4.3% novelty, severity 3 -> reward = 0.075 token[131]
```

and an explanation as above.

Smart Contract Deployment & Usage Summary: - Deployed on Ethereum-like blockchain (likely testnet for dev, mainnet or sidechain for production). - Owned by the project owner, who can assign `minter` role to the detection bots. - Each detection triggers `submit_data_block`, which mints SDT tokens: 95% to the reporter (which is the bot's designated recipient address) and 5% to the treasury. - The event is logged for anyone to see, containing a hash of the data and the key metrics (so one could roughly gauge the event severity from anomaly_bps and severity). - Over time, the total supply of SDT increases with each event. `total_supply` can be fetched (in test after one event it increased by 0.075 token[131][154]). - If needed, the token can have value or governance rights.

The integration with Python code is straightforward via web3 as shown: one just needs the contract instance. In production, one would skip recompiling the contract (already deployed) and instead load it via ABI and known address:

```
from web3 import Web3
w3 = Web3(Web3.HTTPProvider("<RPC_URL>"))
abi = json.load(open("SentinelDataToken.abi.json"))
token = w3.eth.contract(address="<deployed_address>", abi=abi)
```

```
# Assume w3.eth.account is set up with private key for minter
tx = token.functions.submit_data_block(hash, anomaly, novelty, severity,
recipient).build_transaction({...})
# sign and send...
```

This would integrate into the Python backend whenever an alert qualifies.

Vyper and Web3 Dependencies: They needed Vyper 0.3.10 to compile the contract (we saw pip install logs)[155][156]. In practice, once deployed, you don't need Vyper at runtime, just the ABI. Web3.py and eth_account are needed to send TXs.

Attribution and credit in contract: The token name "SentinelData" and symbol "SDT" indicate they treat it as a specific utility token for the sentinel data system. Possibly crediting the author (Anirudh) not directly on chain except maybe he's owner.

In conclusion, the smart contract integration closes the loop by: 1. Committing the detected event to an immutable ledger. 2. Emitting a reward token that quantifies the contribution of anomaly detection (which could eventually be traded or used). 3. Enabling decentralized verification and potentially participation in the depeg monitoring process.

This kind of on-chain bridge is fairly cutting-edge for ML systems, blending off-chain analysis with on-chain incentives – making the Sentinel not just a monitoring tool but part of a larger ecosystem of DeFi risk management.

11. Retraining and Drift Handling

To keep the ML system accurate over time, the Depeg Sentinel incorporates automatic **retraining** and drift mitigation procedures. This ensures the models and detectors adapt to evolving data distributions (for example, if a stablecoin changes its behavior or market conditions shift) and continue to perform well.

When to Retrain: The logic for deciding retraining is twofold: - **Scheduled Retraining:** By default, the system is configured to retrain models **nightly** (once per day). This is a conservative approach to ensure the models get updated regularly with the latest data, even if no obvious drift is detected. The meta-policy's retrain_check always returns `should_retrain: True` with reason "scheduled nightly" if drift isn't triggered[35]. In practice, one might schedule this at a low-activity time (e.g. midnight UTC). - **Drift-Triggered Retraining:** If the **feature drift monitor** detects significant drift in the data distribution, it triggers retraining immediately. The drift check uses KS-test and PSI metrics as described in Section 2 and 8. If any key feature's KS or PSI exceeds threshold (e.g. KS ≥ 0.2 or PSI ≥ 0.25)[30][32], `retrain_check` will flag drift[33][34]. This suggests the model's training distribution is no longer representative of current data, so the model may be making errors. In that case, `should_retrain: True` with reason "feature drift" is returned immediately, prompting a retrain sooner than the nightly schedule.

Additionally, the comment "drift threshold (PR-AUC drop) hit" in code[35] implies they intended to consider model performance degradation (like if the detectors' PR-AUC or forecasters' AP falls below some threshold) as a trigger. That isn't explicitly implemented except perhaps indirectly by always retraining daily. But a more advanced setup could periodically evaluate model on rolling window and if AP/Brier worsens beyond tolerance, retrain.

Retraining Process: When a retrain is triggered (either by schedule or drift): 1. The system collects the latest data from the `live_dataset.csv` (or internal DataFrame). It likely uses as much recent data as available (maybe up to a certain limit or window of days) to train the forecasters. 2. It recomputes or updates the **labels** for the forecasters. Using `_ensure_labels` ensures we have a `y_10m` and `y_30m`

column in the live data, adjusting thresholds if needed[48][157]. If previously some data was labeled, it may append new labels for new rows. They also handle the case if all labels are negative (it will try lower thresholds to get positives)[14], or if still no positives it picks top anomalies as pseudo positives[15]. - Note: They only label if `y_10m` not in df or is all one class, indicating not labeled or label issue[158][159]. 3. **Train new forecaster models:** They instantiate new XGBoost models and fit on the updated dataset (`train_forecaster_10m()` and similarly maybe `train_forecaster_30m()`). They select the feature columns (ensuring the chosen features exist in df; the code dynamically picks those present)[9][160]. After training, they likely save the model (perhaps in memory or disk using joblib or XGBoost's save). 4. **Calibrate new models:** Once new models are trained, they generate new calibration curves. The nightly process calls `save_all_calibration_artifacts()` which runs calibration for 10m and 30m models and saves JSON and PNG plots[20][55]. Isotonic regression is applied on the model's validation set predictions as described earlier. The new calibration files (calibration_10m.json/png, calibration_30m.json/png) are written to artifacts. 5. **Evaluate detectors:** The nightly retrain routine also evaluates and saves detector performance (via `save_detector_pr_auc()`)[20][45]. This likely retrains unsupervised detectors? Actually, unsupervised anomalies like IF or OCSVM might be re-fit on new normal data distribution. The code `save_detector_pr_auc(thr_abs_dev=0.003)` reads the current data's dev to label anomalies >=0.3% and computes PR-AUC for each detector's output vs that label[45][161]. It returns a dict with scores and a winner, which they save in `detector_pr_auc.json` and a PNG chart. - If the distribution changed, they might also **retrain detectors** that require fitting (IF, OCSVM, AE). It's not explicitly shown, but likely done elsewhere (maybe inside `run_anomaly_zoo_update_live()` which could refit IF/OCSVM on the latest portion of data flagged as normal). The snippet `sample_once(); run_anomaly_zoo_update_live()` in main_demo suggests an initial training of detectors on some baseline sample[162][163]. For periodic retrain, they might similarly call a function to refit detectors on a sliding window (maybe last N days). - E.g., Isolation Forest might be refit on the last X hours of data labeled as normal (not containing a big depeg). - Autoencoders could be retrained or fine-tuned on new data if needed. - Because unsupervised detectors can degrade if distribution shifts significantly (e.g., if a stablecoin changes volatility regime, the LOF's notion of "normal" density should be updated). - The presence of drift alert itself might encourage refitting them.

1. **Recompute severity model:** If retraining is triggered due to new type of events, one might also want to update the severity calibrator or model:
2. `train_severity_calibrator()` will be called in nightly routine (the code snippet suggests severity calibrator is trained using the EVENTS_JSON events list[164][65]). If new events were added, it retrains the calibration mapping for severity (the underlying SEV_MODEL might not retrain unless it's an ML model; it could be a static language model and logistic that might be updated if new training data available).
3. They update events JSON with any new incidents and then call `enrich_events_with_calibrated_severity()` to recalibrate severities in the list[165][72].

4. **Update Meta Info:** After retraining, the system logs it by writing `RUN_META.json` with current timestamp, versions of libs, config values (like pools monitored), artifacts paths, and some counts (live_rows, number of pools)[166][167]. This file acts as a snapshot of this run's state. It might also include if extra info was provided; the `write_run_meta(extra=...)` can merge additional data (like maybe performance metrics or outcome of drift)[168][169].

5. The example `RUN_META.json` after a run shows it recorded ts, versions (python 3.12.7, numpy 2.1.3, etc), config (eth_rpc, mock_mode, lookback period, list of pools), artifact file paths, and counts (378 live_rows, 6 pools)[167][148]. This is useful for tracking which data and version was in use at any time.

6. **Model Snapshotting:** Optionally, they might snapshot the models themselves. Perhaps they save the XGBoost model to disk (like forecast_10m.model or as part of parquets?). Or they rely on re-training quickly so explicit saving is not needed. But since they calibrate after training, they likely keep the model in memory for scoring new data and also save the forecast predictions in a file (forecast_10m.parquet) for the UI.

7. **Handling Running System:** During retraining, one might pause or handle carefully the live scoring. Possibly, they might run training in a separate process or thread, then atomically swap out the model. The code doesn't explicitly mention concurrency issues. Given the data frequency might be low (minute or more), retraining can be done quickly and replacement won't drastically affect real-time performance. The sentinel likely just does it sequentially: e.g., at 00:00, stop, retrain, then continue with new model.

8. **Post-Retrain Drift Reset:** After retraining on new data, the drift metrics will likely reset to false (since now training distribution is updated). The drift check function uses a 70/30 split on the current live data[49][18] – after retrain, presumably it might clear or the effect of drift is less. They also might update some baseline distribution for drift. Possibly, one improvement could be to maintain a reference distribution of features from training set and compare current data to that. But if they always use the last 70% vs last 30%, it's somewhat self-adaptive – drift will usually measure differences between older portion vs newest portion of data. After retrain, the older portion now includes data up to near present, so drift resets until new changes come.

9. **Which Files Are Regenerated:**

    - **Forecast model artifacts:** The main "artifact" of a model retrain is the internal model (not a file visible externally, but they output forecast probabilities to a Parquet file used by UI, and possibly could output model feature importances or shap).
    - **Calibration JSONs and PNGs:** calibration_10m.json, calibration_30m.json, and their plots are updated with new calibration curves[45][55].
    - **detector_pr_auc.json & .png:** updated with latest performance and winner detector[45][21].
    - **feature_drift.json:** likely updated after drift computation; the nightly report mentions JSON: feature_drift.json with metrics[170][171]. In drift detection code, they write out `DRIFT_JSON.write_text(json.dumps(payload))` each time compute_feature_drift is called[172]. So after retraining, possibly drift was measured and feature_drift.json updated with metrics (drift may be false now).
    - **RUN_META.json:** updated each run to reflect new model and environment state[166][167].
    - **Events.json:** possibly appended with any new incidents that were confirmed, and updated severities after calibrator run.
    - **Explainability files:** If new models are in place, they might regenerate explain.json (via explain_forecast_10m/30m on the new model). The nightly routine does `exp30 = explain_forecast_30m(...)` and prints/possibly saves it[74][173]. We might expect `explain.json` for 10m and `explain_30m.json` for 30m updated after retrain so that explanation reflects the new model's behavior.

Logging Snapshots and Meta Info: The system logs meta info on each run (including after retrain) for audit. For example, `RUN_META.json` after retraining includes "pools": [...] listing current monitored pools, which might get updated if, say, they add or remove a pool from config. If `lookback` or `mock_mode` changed, that's recorded. Also library versions – helpful to see if model changed due to library updates. It also shows counts like live_rows: 378 (so you know dataset size)[174].

Additionally, each nightly run exports a **Nightly Model Report** (markdown and PDF) which includes the winner detector, calibration results, drift status, and incidents summary[45][20]. This is more for developer/analyst consumption, but it's effectively a snapshot of model performance and data state at that time. For example:

```
## Winner Detector Today
- Winner: `z_if` (PR-AUC: 0.704)
- JSON: detector_pr_auc.json  PNG: detector_pr_auc.png

## Forecast Calibration
- 10m -> JSON: calibration_10m.json  PNG: calibration_10m.png
- 30m -> JSON: calibration_30m.json  PNG: calibration_30m.png

## Feature Drift
- drift: **True**  reason: feature drift
- JSON: feature_drift.json

## Incidents (last window)
- None.
```

This report (in `report.pdf`) is an artifact that can be saved each night (perhaps with a date in name). It shows if the system needed retraining due to drift (in example, drift was True, so presumably a retrain was done or at least recommended). It also documents model calibration quality each day.

Updating Data Sources: Retraining could also involve updating how data is processed if needed. For example, if a new feature becomes relevant or an existing feature becomes unreliable: - If adding a new data source/feature (like maybe integrating a new oracle or new metric), one would incorporate it into the pipeline (code to fetch, add to live CSV, train model using it). The structure is flexible as they dynamically select features present. If a feature column appears in data, model training will include it (they do `feature_cols = [list]` then `use_cols = [c for c in feature_cols if in df]`[9]). - If removing a feature (like if a data source fails or is no longer useful), just leaving it out of data means model will skip it due to that conditional selection. So the pipeline can handle missing features gracefully to some extent.

Model Versioning and Persistence: - It appears the system does not version model files explicitly; it retrains in place and updates artifacts. If one wanted to keep old models, you could timestamp model files. But RUN_META and nightly report partly serve as historical record. - The question of "reproducibility" is partly addressed by logging and by seeds (maybe they set seeds for XGBoost or rely on large data to not worry about randomness). Not explicitly shown.

Conclusion of Retraining Handling: The Sentinel's retraining strategy ensures it remains **self-calibrating**: - Frequent (daily) refresh to incorporate latest data and maintain accuracy. - On-demand retraining when distribution shifts, preventing prolonged periods of poor performance. - All the while, storing relevant metadata and artifacts so researchers can see what changed (drift metrics, new calibration curves, etc).

From a maintenance perspective, one must simply ensure the scheduled retrain job (maybe a cron or just the code's infinite loop triggers at set time) is running. The user can also force retraining by hitting an API or toggling something (maybe calling `/policy/retrain_check` returns always true so that doesn't need force, one could just run the train pipeline any time).

Potential CLI for Retraining: They might have a CLI entry or function `train_if_needed(force=True)` to trigger it manually. The snippet shows `train_if_needed(force=False, min_hours=4.0,`

`min_new_rows=400, label_drift_thr=0.15`) being called in a loop[87]. Possibly: - `min_hours=4.0`: it won't retrain more often than every 4 hours. - `min_new_rows=400`: need at least 400 new data points since last train. - `label_drift_thr=0.15`: maybe if calibration degrade or something (not sure). - If those conditions met or `force=True`, then it will retrain (calls the steps above). - This suggests the live pipeline possibly retrains not exactly nightly but as soon as conditions satisfied (like each 4h if data is streaming fast, or at least every 400 points). But the meta-policy indicated daily anyway.

Thus, retraining is robustly integrated, requiring minimal human intervention. If the stablecoin market undergoes structural change, the sentinel should adapt within a day at most, often faster if drift is obvious, thereby keeping false alarms low and detection sensitivity high.

12. Data Sources and Ingestion

The Sentinel's analysis is only as good as the data it ingests. It uses a variety of **data sources** related to stablecoin liquidity pools and prices. Below we detail what data is collected, how it's processed, and what assumptions are made about it.

Monitored Pools and Metrics: By default, the configuration (as seen in `RUN_META`) included pools: - `USDC/USDT_univ3` – a Uniswap V3 pool for USDC-USDT, presumably on Ethereum (or another chain if specified). Uniswap V3 pools provide on-chain price and liquidity info. - `DAI/USDC_univ3` – a Uniswap V3 pool for DAI-USDC. - `3pool_curve` – the Curve 3pool (USDC, USDT, DAI) on Curve Finance, a stablecoin pool. These cover major stablecoin pair venues. The system can be extended to any pool (just need the addresses and possibly ABI info to query them).

For each pool, the sentinel likely gathers: - **Pool Price (DEX price):** The price of one stablecoin in terms of the other in the pool. For Uniswap V3, the price can be derived from the `sqrtPriceX96` value in the pool's slot0 state[3][4]. The code snippet shows: - They fetch `slot0().sqrtPriceX96` and tokens' decimals to compute price[175][4]. - The formula: `p_raw = (sqrtP / 2**96)^2 * 10^(d0 - d1)` yields the price of token0 in token1 (or vice versa depending how they treat). - They also fetch token0 and token1 addresses, and their decimals and symbols (to know which assets)[176][177]. - In stablecoin pools, the price should be near 1.0. So dev = price - 1.0 effectively (if price of USDC in USDT is 1.002, dev ~+0.002). - **TWAP or Oracle Price:** The system references external price or TWAP for context: - It uses Chainlink or similar oracles. In code, `get_chainlink_price(aggregator_address)` returns a price with updated timestamp[178][179]. They call aggregator's `latestRoundData()` and get answer and last update time[178]. That provides an **oracle price** (often considered ground truth stablecoin price in USD). - `spot_twap_gap_bps` likely is the difference between the current pool price and a longer-term TWAP. Possibly they compute a TWAP either from the pool (some pools have built-in or they can average historical data) or use the Chainlink as a proxy for TWAP (Chainlink is like a slowly updating reference). If Chainlink for USDC is $1.00 and Uniswap pool is at $0.98, then spot_twap_gap = -2% (-200 bps). - Alternatively, they might compute TWAP from their own recorded `dev` over a window (like average of last hour's price). - **Reserves / Liquidity (TVL):** They likely fetch the total liquidity in the pool: - For Uniswap V3, each pool doesn't explicitly track total TVL easily, because liquidity is distributed by ticks, but you can approximate by summing token0 and token1 reserves at current price. Perhaps simpler is to fetch the pool's token balances or simulate a small swap to get amount out. However, it's easier for stable pools: if price ~1, total TVL ~ sum of both reserves. - For Curve 3pool, the contract provides reserves (via `balances` array or by reading underlying LP supply etc.). Possibly they used the subgraph or some direct query. - They might not explicitly get TVL in code snippet, but `tvl_outflow_rate` suggests they track changes in TVL. How? Possibly they derive it from difference in reserves or from events. It might be approximated by the volume of net withdrawals: `tvl_outflow_rate` could be computed as (TVL_prev - TVL_now) / TVL_prev per unit time (a negative if outflow). Or simply the change normalized by time. If

positive (inflow), outflow rate could be 0 or negative. They probably compute it from the live dataset by looking at differences: e.g. they have a code `r0_delta, r1_delta` which likely are changes in token reserves since last sample[180]. Using those, one could compute how much total value left or entered (valuing each token properly). - **Deviation (dev):** This is a critical feature – how far off the expected 1:1 peg the pool price is. They likely define `dev = pool_price - 1` (if measuring in quote of one stable to another). For multi-asset (3pool), how to define dev? Possibly pick one asset as baseline (maybe USDC) and measure others relative. Or they compute something like the pool's virtual price vs pegged $1. If 3pool is balanced, each token is roughly equal value; they might not directly use 3pool price but something like if one coin is being priced lower by pool vs others. - Possibly they use a synthetic price: e.g. how many USDC for 1 DAI via the pool (should be 1 when equal). - **Rolling std (`dev_roll_std`):** They likely maintain a rolling standard deviation of `dev` over a recent window (maybe last 30 min or 1h). This measures volatility – stablecoins normally have low volatility (std ~0.0001), so an increase indicates instability. - **Spread or Ratio metrics:** - `oracle_ratio`: The ratio of pool price to oracle price, as mentioned. If oracles says USDC=1.00 and pool says 0.98, oracle_ratio = 0.98 (or 98%). - This is slightly redundant to dev and spot_twap_gap, but oracle_ratio near 1 means pool and oracle agree. A departure might indicate either a lagging oracle or a genuine move. - **Freshness flags:** `feeds_fresh` in policy indicates if data is recent. They set this likely by comparing the `updatedAt` from chainlink or on-chain block time vs now. If oracle or pool data hasn't updated (perhaps node is not syncing or chain stalled), they may treat it as stale. - **Cross-Pool features:** from network, `neighbor_max_dev` and `neighbor_avg_anom` (discussed earlier). These rely on having multiple pools data concurrently: - They maintain `_hist_dev` and `_hist_anom` dictionaries for each pool to compute these features[181][182]. - **External data (News/Twitter/etc):** The question didn't mention but sometimes severity might use external news. Not sure if integrated, but since citations are mentioned in snapshot, they might have capability to pull news articles via APIs (maybe `update_events_from_sources()` in code[183][184]). - If an event is detected, they might search a knowledge base or API (maybe a HuggingFace model if `HF_TOKEN` was set as we see huggingface token usage[185]). Possibly to get context or cause of anomaly (like scanning Twitter for "USDC depeg"). - They had a variable `use_rag` (Retrieval-Augmented Generation?) in `main_demo_next_steps`[183]. That hints they might fetch related info from a knowledge base to include in analyst note, and citations pointing to it. This is advanced and not fully elaborated, but it aligns with capturing novelty/cause.

Data Ingestion Implementation: - Likely a separate module or class handles on-chain data fetch. The snippet shows a `DataFetcher` with methods like `get_chainlink_price` and `get_univ3_price` presumably inside some class (the code snippet's context suggests an object with `self.w3` and `self._call`)[186][187]. - They possibly use `web3` to call contract addresses (for Uniswap and Chainlink). For Chainlink, they might have aggregator addresses stored in config for each asset's USD price feed. - For Uniswap, they need pool addresses for the specific pairs. Those can be configured (maybe `CFG.pools` contains addresses; indeed `RUN_META` shows "pools": list of names, but likely somewhere there's mapping from name to addresses). - For Curve, possibly a simpler approach is to rely on the Curve subgraph or their API to fetch virtual price or pool state. If not, they could call the Curve pool contract to get balances and amplification, then derive price for a small trade. But since difference between stable in stable pool is slight, maybe they treat 3pool's "price" as relative supply difference between one coin and others. - Or they might not use 3pool price directly for dev; instead they treat each pair: DAI vs USDC in 3pool, USDT vs USDC, etc. - Another approach: 3pool has an invariant; but a simpler way: any imbalance yields a price difference in swapping. Possibly they approximate dev by (balance_USDC - balance_DAI)/balance_tot * ... hmm not straightforward. They might skip precise, or use Chainlink to measure each stable vs USD and compute differences (like if DAI is $1.01 externally but $1 in pool, novelty). - They tag each data point with a timestamp (likely in ISO string) and pool name.

Assumptions on Data: - **Data Frequency:** Likely sampling every few minutes (maybe every 1 minute or 5 minutes). Uniswap on-chain data can be fetched at will; Chainlink updates typically every few minutes or on price moves. They likely poll at a fixed interval (the schedule in main loop might sample once per minute: see `sample_once()` being called in main_demo and then maybe repeatedly in main loop). - **Time Synchronization:** They log `ts` as `datetime.now(timezone.utc).isoformat()` for when data is fetched[188][189]. They trust local clock for labeling times. For on-chain data, block timestamps could also be used but here probably not needed. - **Data quality:** If a query fails (e.g., RPC issue), they likely handle exceptions and could skip or mark `feeds_fresh=false`. The code returns OnchainResult objects with ok bool and error message if failure[178][190]. If a fetch fails, they might reuse last data or drop that sample. Ideally the system should handle missing data gracefully (e.g., if a price fetch fails for one pool, perhaps skip anomaly calc for that tick). - **Normalization:** They ensure all price metrics are normalized around 1 for stablecoins. They feed raw `dev` in model, which is already a difference from 1. `anom_fused` is between 0 and 1. Other features like `tvl_outflow_rate` might be a raw fraction or per minute percentage. Probably small values as well (like 0.01 = 1% outflow per min). - **Feature Engineering Summary:** - **dev** – numeric (close to 0 normally, positive or negative). - **dev_roll_std** – small number representing recent volatility. - **tvl_outflow_rate** – could be fraction per minute or per hour (like 0.1 meaning 10% out in last hour? might need clarity). - **spot_twap_gap_bps** – e.g. 10 means current price 0.1% above TWAP. - **oracle_ratio** – dimensionless ratio around 1 (should be 1 if consistent). - **anom_fused** – 0 to 1 (treated as probability). - **r0_delta, r1_delta** – raw changes in reserves; might be large numbers (like in units of tokens). They might be scaled or left raw (if raw, tree model might handle or they may scale them relative to pool size). - **event_severity_max_24h, event_count_24h** – integers (severity 1-5, count as int). event_severity_max might be value like 3, event_count maybe up to some number if many events.

Data ingestion pipeline structure: Potentially: - A separate thread or schedule calls a function that queries all pools for latest data (pool price & reserves, chainlink oracle). - The results are assembled into a dict or DataFrame row and appended to `live_dataset.csv`. - The anomaly detectors then run on that new row to compute `z_if ... z_ae_seq`, and `anom_fused`. - That row is now fully populated with features and scores, and then used by forecasters to output `p_10m, p_30m` (which might be written to forecast file and also appended to `live_dataset` or separate predictions DF). - This triggers policy check: if `p_10m` or anomaly indicates issue, do snapshot and possibly contract call. - Then loop sleeps until next sample time.

They likely decouple sampling and inference frequency – maybe they run anomaly update more often than forecast? But probably same frequency is fine here.

Pre-processing steps: - There might be some smoothing: e.g., if chain data has jitter, maybe they do a small rolling average for dev for stability (though then dev_roll_std should measure the unsmoothed). - Converting units: they had to convert raw on-chain values to floats (Chainlink answer often scaled by 1e8 or 1e6, Uniswap sqrtPriceX96 trick). - Ensuring no divide by zero: if a pool had zero liquidity (shouldn't happen in active pool), or if decimals differ (they handled decimals in formula). - `np.nan` handling: Their code uses `fillna(0.0)` for features when training and scoring[191]. This implies if any feature is missing (maybe due to no data), they substitute 0. This could be dangerous (0 dev means no dev, fine; 0 tvl_outflow means no outflow, fine; 0 fused anomaly means no anomaly, which could be okay if data missing triggers detectors also output 0 or NaN replaced 0). - Outlier clipping: Possibly if some feature goes extreme, the tree can handle it, or they might clip dev to e.g. [-0.5, 0.5] because beyond that stablecoin likely broken or out-of-scope. Not explicitly done though.

External Data Assumptions: - *Curve's complexity:* Possibly simplified. Or they might not heavily rely on 3pool's internal price, and mainly use that pool's imbalances via neighbor features. - *Time alignment:*

They assume all pools data points align on the same timestamps when they sample. So if they sample at 12:00, they fetch all pools' state at that block (which if using same RPC call block number, can get nearly same block for all – but they possibly just sequentially call which could be a few seconds apart, likely negligible). - *Feature uniqueness:* Many features are derived from similar underlying values (like dev vs oracle_ratio vs spot_twap_gap are all related). The model can handle correlation, but must be careful of multicollinearity. However, XGBoost can still work with redundant features, albeit might waste splits. The feature importance might show only one being used (as we saw, dev dominated, others fell to 0 importance). - *Chain interactions:* They use Web3 and likely run on an archive or at least a full node for reading. For Uniswap V3, you need either an RPC that supports `slot0` call (which all do) and the pool contract address and ABIs for pool and tokens (they inlined minimal ABI calls as shown for decimals and symbol)[3][192]. - *Oracle trust:* The system assumes Chainlink or whatever oracle used is correct baseline. If an oracle lags or fails (e.g., the infamous delay), the sentinel might interpret a difference as anomaly whereas it might be oracle behind. However, that's fine – a big oracle lag could itself cause issues (traders might exploit it). The `feeds_fresh` parameter might catch if oracle didn't update for too long (meaning stale). - *IDs and mapping:* `pools.keys()` in config likely maps pool name to addresses and aggregator addresses. Possibly stored in a config file or environment.

Data ingestion summary: The sentinel integrates on-chain data (DEX pool state, oracle prices) in real-time, computing features that represent market conditions critical for stablecoin pegs: - Price deviations and spreads to external anchors, - Liquidity flows and volatility, - Multi-pool relationships (to differentiate isolated incidents vs market-wide moves). This data pipeline runs continuously, feeding the ML detectors and predictors.

Before moving on, ensure any needed environment details: The user should set environment variables or config: - `ETH_RPC_URL` for web3 to connect (unless using local geth or Infura). - `PRIVATE_KEY` for contract calls. - Possibly addresses for Uniswap pools and chainlink feeds (maybe pre-coded). - The code had `CFG.lookback` and `CFG.pools` presumably loaded from a config file or .env.

Given the context, the system likely uses a `.env` file or config JSON specifying pool addresses and feed addresses for each pool name. Developer must update that when adding new pools.

13. Developer Setup and Deployment

For developers or researchers setting up the Depeg Sentinel, here are the steps and requirements:

Environment and Dependencies: - **Python Version:** The codebase requires Python 3.10 or newer (observing features like type hints using | for Union, which is Python 3.10+). - **Package Requirements:** Key dependencies include: - **numpy** (for numeric computations), - **pandas** (for data manipulation, CSV I/O), - **scikit-learn** (for IsolationForest, LOF, OCSVM, IsotonicRegression, LogisticRegression, etc.), - **xgboost** (for the gradient boosting forecaster), - **scipy/statsmodels** (for KS tests, PSI maybe implemented or manually coded, statsmodels was imported possibly for KS if not manual[92]), - **web3.py** (for interacting with Ethereum nodes and contracts), - **eth-account** and **hexbytes** (often needed by web3), - **vyper** (only needed if you plan to compile/deploy the contract yourself; not needed if contract already deployed and ABI is present), - **requests** (for making any external calls or posting alerts), - **streamlit** (for the dashboard UI), - **plotly** (for interactive charts in dashboard), - **streamlit_autorefresh** (for auto-refresh utility), - **pytest** (the code references some tests, and a `monkeypatch` fixture for demo mode), - **sentence-transformers** (for severity model text embeddings), - **torch** (since sentence-transformers uses PyTorch), - **pydantic** (was seen in logs likely pulled in via web3), - Possibly **sklearn-contribs** if they used any special metrics (but likely not). - **Installation:** these can be installed via pip. A `requirements.txt` might be provided. If not:

```
pip install numpy pandas scikit-learn xgboost web3 eth-account vyper==0.3.10 requests
streamlit plotly streamlit-autorefresh sentence-transformers
```

(Note: vyper requires specific version; if not doing contract compile, not necessary to install it in production environment). - **File Layout:** The repository might be structured like:

```
config.py or config.json (pool addresses, etc)
sentinel.py (main pipeline code)
detectors.py (maybe with detector logic or custom wrappers)
forecaster.py (training and inference code)
dashboard.py (the Streamlit app)
contracts/ (with SentinelDataToken.vy and compiled ABI)
artifacts/ (output directory created at runtime)
tests/ (if any notebooks or test scripts)
```

From user files, it seems code might be in Jupyter notebooks (they had .pdf printouts of "Contract", "App", etc). But presumably for production it's turned into .py modules. - **Configuration:** likely via environment variables: - `WEB3_RPC_URL` – Ethereum RPC node endpoint. - `PRIVATE_KEY` – the private key for the minter address (in hex string format). - `HMAC_SECRET`, `API_KEY` – for securing API endpoints (the code uses HMAC with a secret to sign requests to internal API calls[193][23]). - There's mention of `X-API-Key` and `X-Signature` headers in API usage, implying the REST endpoints are protected by an HMAC auth scheme. The `HMAC_SECRET` is used to sign timestamp+body with SHA256[194][80]. So the user should set `HMAC_SECRET` (and `API_KEY` maybe to a known value) for the server and use same to call endpoints. - `OUT` – output directory path for artifacts (if not default). They used `OUT` env to find where to write files[93]. If not set, defaults to a relative path (maybe "outputs/" or "artifacts/"). - `POOLS` or a config file path – to specify which pools to monitor. Possibly they encode that in code or environment variables (like comma-separated list or a JSON). - `MOCK_MODE` – set to "1" to run in demo mode without real web3 (the test uses monkeypatch to set this[96]). In demo mode, the system doesn't actually hit web3 or contract, it generates dummy data. For development without chain connection, one can run with MOCK_MODE=1 to simulate. - `HF_TOKEN` – HuggingFace API token if using any RAG or external model for context, as seen in secrets printing[185]. - **Running the Backend:** Assuming all config is set: - If the code is a script, simply run `python sentinel.py`. This likely starts the loop: fetch data, update anomalies, make predictions, check policy, etc. Alternatively, it might be an API server (if they used FastAPI for `/ml/score_zoo` etc, then you run `uvicorn main:app`). - The presence of `openapi.json` suggests they used FastAPI to expose certain endpoints (score_zoo, healthz, etc) so one might run the backend as an API server. Possibly `uvicorn sentinel:app --reload`. - However, since they also do internal scheduling, maybe they run the FastAPI in one thread and also have background tasks for sampling etc (FastAPI supports background tasks or `while True: ...` with `asyncio`). - The endpoints like `/ml/score_zoo` and `/policy/decide` allow external triggers or integration (like a UI or external orchestrator could call them to get data or force actions). - The simpler approach: The code might not rely on external calls; it could run as a self-contained loop and just use the API for additional insight or to integrate with other systems. - The `healthz` and `readyz` endpoints show they intended to run it as a service for monitoring. - The first time, it will compile the contract (if needed) or load contract, possibly deploy if not address given (the code compiled and deployed to EthereumTester if no PRIVATE_KEY, meaning local dev). - It will then begin sampling data. If mocking, it will generate synthetic data. If live, it will poll on-chain. - One should see output logs in console (they print various statuses: [ok] imported modules, [rpc] connected, [deploy] address, [zoo] anomaly scores updated, [alert] posted alerts, etc as in logs). - The developer can monitor log output or check artifacts to see things like detector_pr_auc.json being produced, events.json logging incidents, etc. - For contract integration, ensure the account has ETH to pay gas if on a real network, and the contract address is configured (the code as is deploys a new

contract on startup if none given, but in production you'd deploy once and provide the address and minter key).

- **Running the Dashboard:** With the backend running (or in demo mode, not strictly needed), run:

  ```
  streamlit run dashboard.py
  ```

  This opens the UI at localhost:8501. In the sidebar, ensure `Demo Mode` is off if you want live data. The UI will read from the artifacts directory (`OUT_DIR`). So it's crucial the `OUT_DIR` env for dashboard matches where backend writes. If running on same machine with default, should match. If not, adjust `OUT` env for the dashboard process. If using separate environment or container, might need to mount a volume for artifacts or have the dashboard fetch data via the API instead of direct file access (not currently how it's built, it reads local files).

- **CLI Entrypoints or Scripts:** Possibly there are scripts or CLI commands:

- e.g., `python sentinel.py --retrain-now` could trigger immediate retrain (not sure if implemented, but the building blocks are there; one could call the internal functions).
- Or `python sentinel.py --once` might just do one sample and exit (for testing).
- If packaged, they might provide a console script like `sentinel-server` to run the API server and `sentinel-dashboard` for UI, etc.

- **License and Attribution:** The documentation should include a note that the project is released under the MIT License. That means the code is free to use and modify with proper attribution. We should mention:

- The original author: Anirudh Reddy Ninganpally, and possibly any contributors or affiliated organization if any.
- A statement like "This project is licensed under the MIT License – see the LICENSE file for details. You are free to use, modify, and distribute this software with attribution. If used in academic or research work, please acknowledge the author."
- This encourages reusability and academic collaboration, aligning with the research-grade nature of the system.

- **Deployment considerations:**

- For production, one might deploy the backend on a server with reliable Ethereum node access. Possibly Dockerize it, making sure to include web3 and model dependencies. Running it continuously requires stable network connectivity to Ethereum.
- The smart contract would ideally be deployed on a known network (e.g., Ethereum mainnet, or a specific testnet for testing).
- The dashboard can be run on a server or in Streamlit sharing. If multi-user or internet facing, one might add authentication because it might show sensitive info (private key is not exposed, but it could allow triggering contract if endpoints unsecured).

- **Testing:** The presence of tests suggests one can run `pytest` to run unit/integration tests. Some tests might spin up an eth tester (as we saw in logs) and simulate a scenario end-to-end, verifying output artifacts.

Repository Structure and Documenting Code: Ideally, the documentation should mention where each part is: - The architecture diagram image can be placed in `docs/` or embedded as above. - Each module

(detectors, forecasting, policy, etc) should have docstrings describing its purpose so researchers can follow easily. - If posting on GitHub Pages (say, using mkdocs or Sphinx), ensure the Markdown is compatible. The instructions specified "Markdown-based and compatible with Sphinx or GitHub Pages", so likely they'd include this in a docs site.

In conclusion, setting up Depeg Sentinel requires a modern Python environment with data science and blockchain libraries, some configuration for chain access, and understanding of how to run both the backend and front-end. Once configured, it provides a near turn-key solution for monitoring stablecoin peg health, which can be invaluable for researchers or DeFi operators. The open-source MIT license encourages others to build upon this system, add more pools, refine models, or integrate it into broader risk frameworks, with appropriate credit to its creator Anirudh Reddy Ninganpally.

14. Attribution and License

DeFi Depeg Sentinel was created by **Anirudh Reddy Ninganpally** (the primary author of this project). The project is released under the **MIT License**, a permissive open-source license. This means that anyone is free to use, copy, modify, merge, publish, distribute, and sublicense the software, provided that the original copyright notice and permission notice (the MIT License text) are included in all copies or substantial portions of the software.

In practical terms, you can incorporate this system into your own research or products with minimal restrictions. We do encourage giving proper **attribution** to the author when using the code or derivatives of it. For academic use, please cite the project or credit the author in relevant publications. For commercial use, the MIT license allows it, but it’s good practice to acknowledge open-source contributions.

The choice of MIT License reflects the project’s goal of being reusable and extensible for the community. You can modify the system (for example, add new detectors, support new blockchains or off-chain data) and you do not have to open-source your modifications (though contributing improvements back to the community is welcomed). The only requirements are to retain the license notice and not hold the author liable for any issues (standard MIT terms).

A quick summary of the MIT License terms:

```
MIT License

Copyright (c) 2025 Anirudh Reddy Ninganpally

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software... to deal in the Software without restriction, including without limitation the
rights to use, copy, modify, merge, publish, distribute... subject to the following
conditions:
- The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Software.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND...
```

(Refer to the LICENSE file in the repository for the full text.)

By using this software, you also implicitly agree that any tokens (SDT) minted by the system’s smart contract have no warranty of value or security; they are simply a mechanism within the research project.

In conclusion, feel free to experiment with and build upon DeFi Depeg Sentinel. It was designed as a research platform to advance stablecoin risk monitoring, and the MIT license and public release are

meant to catalyze further innovation. Please credit the original author and project where appropriate, and enjoy the benefits of an open and collaborative development model in the DeFi research community.

---

[1] [2] [36] [37] [52] [53] [58] [91] [92] [93] [94] [98] [99] [100] [101] [102] [103] [104] [105] [106] [107] [108] [109] [110] [111] [112] [113] [114] [115] [116] [117] [118] [119] [120] [125] dashboard.py

file://file-TT7WByx5dZoM63Dc1ctTrP

[3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [54] [55] [56] [57] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [74] [75] [76] [77] [78] [79] [82] [83] [84] [85] [87] [90] [95] [96] [97] [121] [122] [157] [158] [159] [160] [161] [162] [163] [164] [165] [166] [168] [169] [170] [171] [172] [173] [175] [176] [177] [178] [179] [180] [181] [182] [183] [184] [185] [186] [187] [188] [189] [190] [191] [192] sentinel.pdf

file://file-LPYYAis27PMCt9fxXDi3Fv

[23] [24] [80] [81] [88] [89] [193] [194] contract.md

file://file-5Vv6F5nzT5ZbYfeZfANeqp

[38] [39] [73] [86] [126] [127] [128] [131] [132] [133] [134] [135] [136] [137] [138] [139] [140] [141] [142] [143] [144] [145] [146] [147] [150] [151] [154] Contract.pdf

file://file-AtBVkoiStekiUUsAbG6cH4

[123] [124] App.pdf

file://file-Dht7A1iTyRkwNTMKCfnPg9

[129] [130] [152] [153] SentinelDataToken.abi.json

file://file-7FEMYCEhGXiAC3yWS1nyUe

[148] [149] [167] [174] RUN_META.json

file://file-6pkyY1hEYuRyaNnEuk3qD2

[155] [156] Deployment.pdf

file://file-33n8GvvDtPmwUNREtfMcki