ChatGPT

Closed-Loop AI Feature Learning for Financial Models

Overview

Designing a **closed-feedback-loop system** for feature learning in finance allows AI-generated features (from deep/self-supervised models or symbolic learners) to continually improve using downstream performance signals. In such a loop, features are not static; instead they are **iteratively refined based on backtest results and predictive model performance**. The goal is to automatically discover and enhance predictive signals (often called *alphas* in quantitative finance) by linking feature generation tightly with strategy evaluation. This requires careful architecture design to connect data pipelines, feature engineering modules, modeling, and backtesting in one continuous process. Key strategies include reinforcement learning that treats feature search as a **reward-driven exploration**, differentiable or proxy-based backtesting to propagate performance feedback, online learning to adapt to evolving regimes, and **neuro-symbolic techniques** to distill complex learned features into interpretable forms. We will discuss how to implement this loop, the data flow and retraining cycle, and methods to ensure the discovered features remain robust (avoiding backtest overfitting, generalizing across market regimes, and optimizing for risk-adjusted returns).

Feedback Loop Architecture and Pipeline

A closed-loop feature learning system can be structured as an iterative pipeline with the following stages:

1. **Data Ingestion and Preprocessing:** Market data (price, volume, fundamentals, alternative data, etc.) is continuously collected and cleaned. This can include *self-supervised learning* on unlabeled historical data to initialize raw feature representations (e.g. via autoencoders or contrastive learning that capture structure in time-series data).

2. **Feature Generation Module:** An AI-driven component generates candidate features or signals from the data. This might be a deep neural network that produces latent features, an evolutionary algorithm that formulates mathematical combinations of inputs, or a **reinforcement learning agent** that selects transformations. The feature generator can operate symbolically (creating formula-based features) or through learned embeddings. Initially, it may propose many features or strategies (e.g. technical indicators, statistical features, or patterns learned via self-supervision).

3. **Model Training & Backtesting:** Downstream, a predictive model (or a trading strategy) uses these features and is evaluated on historical data. This involves training a model (for example, a forecasting model or a policy network) on a training period and then **simulating a backtest** on a validation period to compute performance metrics. The backtester computes returns, Sharpe ratio, drawdowns, etc., as if the model's predictions were used in trading. Crucially, this stage provides the **performance feedback signal** for the loop.

4. **Performance Evaluation and Feedback:** The results from backtesting (e.g. validation Sharpe ratio, profit factor, predictive accuracy) are fed back into the feature generation module. This closed loop creates a *feedback signal* that guides the feature generator to improve. For instance, features that led to poor performance can be modified or discarded, while aspects of successful features are amplified or recombined. In an RL context, the backtest metrics serve as the **reward signal** for the feature-search agent. In an evolutionary strategy, the metrics serve as the fitness score for feature candidates.

5. **Retraining / Iteration:** Using the feedback, the system updates the feature generation strategy (e.g. adjusting the RL policy or updating the symbolic regression). New candidate features are generated and the cycle repeats with updated models and backtests. This iterative retraining cycle continues (potentially in an ongoing online fashion or in periodic batches), enabling **continual improvement** of the feature set. The data pipeline continuously brings in new market data so the loop can adapt to recent regime changes.

This architecture requires automation and careful orchestration (often implemented with an MLOps pipeline). Data flows from raw sources to feature generators to backtest systems in a streamlined way, possibly in real-time for intraday systems or in daily/weekly batches for longer-horizon strategies. The closed-loop design ensures that **feature engineering is not a one-off process but an adaptive cycle** driven by actual trading performance.

Reinforcement Learning for Feature Discovery

Reinforcement learning (RL) provides a natural framework for closed-loop feature search by treating the **feature generation process as a sequential decision-making task**. In this approach, an RL agent interacts with an environment where at each episode it proposes a set of features or a trading policy, and then receives a reward based on backtest performance. The reward can be defined as the **strategy's performance metric** (e.g. average return, Sharpe ratio, or any risk-adjusted return from the backtest). The agent's goal is to maximize this reward by evolving better features over time.

One strategy is to define the agent's **action space** as transformations or combinations applied to base signals. For example, the agent could sequentially apply operators (like moving averages, differences, ratios, nonlinear transforms) to build a feature; the episode ends when a feature or feature set is constructed, and the policy is then evaluated via a backtest reward. Over many iterations, the RL agent learns which types of features yield higher rewards. This essentially performs an **automated feature engineering**, guided by the trading outcome. Notably, **genetic programming (GP)** and other evolutionary algorithms can be seen as special cases of this idea, where candidate formulas (features or strategies) are mutated and selected based on a fitness score (e.g. Sharpe ratio). For instance, a GP approach was used to evolve trading rules that directly **maximize the Sharpe ratio**, significantly outperforming linear and even some nonlinear methods by capturing complex patterns [1]. This demonstrates that searching the feature space with an objective linked to performance can yield highly nontrivial solutions that traditional methods miss.

A concrete example in recent research combined RL with symbolic feature discovery. In a meta-reinforcement learning setup for trading, **logical program induction** was used to discover frequent symbolic patterns in price data, and incorporating these human-interpretable features into the RL agent's state improved performance [2]. In other words, an RL agent (using a meta-learning algorithm called *RL^2*)

benefited from **learned symbolic features**, suggesting that RL can effectively leverage and refine features that it discovers. Another study in high-frequency trading introduced an RL agent that **enriched its own feature space** (through an interpolation mechanism) and used a performance-driven reward; the result was an agent that *statistically significantly outperformed* baselines in returns and even achieved **improved risk-adjusted metrics**, all while adapting in real time [3]. These examples illustrate that an RL-based feature search loop can yield features and strategies that are continuously optimized for actual trading success, rather than just predictive accuracy.

Implementation-wise, training an RL agent for feature discovery means designing a reward function carefully. A common choice is to use *backtest profit or Sharpe ratio as the reward*. One must ensure the reward captures the long-term performance (which might involve multi-step decision making if features are sequentially constructed). This can be approached with episodic RL: the agent generates a full feature set, then gets a single reward (the backtest result) at episode end – effectively a black-box optimization via RL. Alternatively, one can give incremental rewards (e.g. intermediate rewards for partial feature performance or after each trading period) to help credit assignment. The closed-loop is achieved as the agent continuously updates its policy for feature generation based on these rewards, thereby **searching the space of features or trading rules** and gravitating towards those that yield good backtest outcomes.

Differentiable Backtesting and Decision-Focused Training

A major challenge in closed-loop training is that **backtesting a trading strategy is not fully differentiable**. Traditional backtests involve simulating discrete decisions (buy/sell/hold) with if-else logic, which breaks the gradient flow. However, recent research in *decision-focused learning* tries to bridge this gap by making the downstream financial metric influence the model training directly. One approach is to create a **differentiable proxy of the backtest** – for example, use continuous position sizes or probabilities instead of hard buy/sell triggers, so that the P&L (profit and loss) becomes a smooth function of model outputs. By doing so, one can in principle compute gradients of performance with respect to model parameters or features. In practice, certain metrics like the Sharpe ratio can be expressed in a differentiable form given a series of returns, enabling gradient-based optimization. Indeed, it has been demonstrated that the **Sharpe ratio can be embedded directly into the loss function of a neural network** and optimized via backpropagation [4]. This end-to-end approach means the model is trained to maximize a financial objective (Sharpe in this case) rather than a prediction error, effectively making the backtest performance the training goal.

That said, directly optimizing a non-linear, non-convex objective like Sharpe or drawdown via gradient descent is tricky – it can lead to unstable training and local optima [5]. To address this, one strategy is using **proxy or differentiable approximations** of the backtest. For example, an approach called *Decision-Focused Learning* might solve a simplified portfolio optimization (such as maximizing Sharpe under certain constraints) to generate "optimal" outputs, and then train a model to predict those outputs via a conventional loss. An example is the *Decision by Supervised Learning (DSL)* framework, which reframes portfolio construction as a supervised learning task: it first computes the optimal portfolio weights that maximize Sharpe/Sortino on the training data, then trains a network to predict those weights using a standard loss (cross-entropy) [6]. This acts as a proxy – the model isn't directly optimizing Sharpe, but it's learning to imitate the decisions that *were* optimal for Sharpe. Such proxy-aware training aligns the model with the final objective while maintaining the stability of using a well-behaved loss function. Another technique is to use **convex differentiable surrogates** for objectives – for instance, using differentiable portfolio optimization layers or custom losses that approximate drawdown or turnover penalties.

In summary, making backtests *differentiable* often involves either (a) **relaxing the trading strategy** into a soft, continuous form so gradients exist, or (b) **bi-level training** where an inner optimization provides targets for an outer learning process. Both approaches close the feedback loop by allowing the performance metrics to directly influence feature/model parameters. The result is a more efficient feedback loop: rather than just trial-and-error (as in pure RL or evolution), the model can **ascend the gradient of a financial reward signal**. For instance, researchers have derived gradients of the Sharpe ratio and used them in policy-gradient algorithms [7] or applied "direct reinforcement" methods that treat **risk-adjusted return** as a reward to optimize (Moody & Saffell's differential Sharpe ratio approach). These techniques effectively allow *backtest performance to act as a training signal* in a closed-loop manner. The caveat is that great care is needed to ensure the differentiable approximation truly correlates with the eventual backtest performance and that the optimization doesn't overfit (a topic we address later).

Continual Learning for Evolving Time-Series

Financial data is non-stationary – market behavior changes over time (regime shifts, new structural breaks, etc.). Thus, a closed-loop feature learning system must incorporate **online or continual learning** so that it adapts to new data and does not become stale. Rather than performing a one-time training on historical data, the system should be designed to **learn continuously from streaming data** or through periodic retraining on expanding windows. This can be achieved via frameworks for *online learning*, *incremental model updates*, or *meta-learning*.

In practice, this means after each new batch of data (e.g. daily or monthly), the system updates both its features and predictive models. Online learning algorithms (like stochastic gradient descent run for each new data point, or specialized online learners like online boosting or Bayesian updates) allow the model to **adapt rapidly while retaining past knowledge**. Continual learning is challenging because of the risk of *catastrophic forgetting* – new patterns might overwrite previously learned ones. Techniques such as experience replay, elastic weight consolidation, or dual-memory systems can be used to balance plasticity and stability. As one recent paper noted, *learning online from a time series requires both high plasticity to adapt to new regimes and stability to not forget recurrent ones* [8]. In fact, researchers have reframed online time-series forecasting as an **online continual learning problem** [8], highlighting the need to explicitly handle evolving distributions.

A closed-loop pipeline can implement continual learning by scheduling regular **retraining cycles**. For example, a rolling window approach retrains the feature generator and model every month using the latest N years of data, always keeping a recent view. More advanced is a truly online update: as each new data point arrives, the feature evaluation loop could run in mini-batch mode – updating model weights slightly and re-evaluating feature performance incrementally. The architecture might maintain a **replay buffer of recent and older data** to ensure the model doesn't completely forget past regimes. Some approaches use **meta-learning** where the model is trained to update itself quickly with new data (so it can handle regime changes with minimal additional training).

The data pipeline in an online setting should incorporate a **feedback mechanism on new data** as well. For instance, if the backtest on recent data shows certain features deteriorating in performance, the system can flag those for replacement or refinement in the next feature generation cycle. Continual learning also implies continuous validation: as new regimes appear (say a volatility regime change), the system might detect concept drift (via statistical tests or monitoring the model's error distribution) and trigger a feature discovery sub-loop to find new signals suitable for the new regime. In summary, **time-series continual**

learning ensures the closed-loop pipeline remains effective through different market conditions by constantly updating and *not freezing the feature set*. The loop, therefore, becomes an ongoing adaptive process, rather than a static train-test deployment.

Distilling Deep Features into Symbolic Forms

AI-generated features from deep learning models are often hard to interpret (e.g. latent vectors or complex nonlinear combinations of inputs). To maintain an effective feedback loop – especially in finance where interpretability and trust are important – it can help to **distill complex features into simpler, symbolic forms**. Distillation here means taking a feature or mapping learned by a deep model and expressing it as an approximate *closed-form formula or rule* that humans can understand. This can create a virtuous cycle: the deep model proposes powerful patterns, and symbolic regression or rule induction translates those into **interpretable features** that can be vetted, stress-tested, or even modified by domain experts before feeding back into the model.

One popular approach is to use **symbolic regression** tools (such as PySR or genetic programming) to find mathematical expressions that replicate a deep model's output. For example, if a neural network is generating a feature $F(t)$ from inputs (perhaps through some multilayer nonlinear transformations), one can generate a dataset of inputs vs. the network's feature output and then run symbolic regression to see if there's a concise formula that maps inputs to $F(t)$. PySR, in particular, has been used for "**symbolic distillation**" of neural networks: essentially converting a neural net into an analytic equation [9]. This yields features in forms like $(\text{EMA}(price, 10) - \text{EMA}(price, 50)) / \sigma(\text{returns})$ or other readable formulas, rather than opaque activations. Importantly, such **neuro-symbolic features** can be both interpretable and potentially more robust. In scientific applications, it's been shown that symbolic models extracted from a trained neural network not only are simpler, but **sometimes generalize better to new data than the original neural net** [10]. In finance, this means a formula distilled from a deep model might maintain performance across market regime changes better than the original model that found it, because the symbolic form often captures the essential structure without overfitting to idiosyncrasies.

After distillation, the **symbolic features** can be fed back into the pipeline in multiple ways. They can be provided as additional inputs to models (enhancing the feature set with these new interpretable signals), or even replace more complex features if they perform comparably. Furthermore, because they are human-readable, portfolio managers and researchers can scrutinize these features, apply economic intuition, and do stress tests (e.g. check how the feature behaved in past crises or under different conditions). This manual oversight can act as an additional feedback filter to prevent the loop from chasing spurious patterns. The closed-loop can thus involve a collaboration between AI and humans: the AI finds a complex pattern, distills it symbolically, and a human or an expert system evaluates its soundness before allowing it as a permanent feature.

Neuro-Symbolic Feature Refinement

Combining neural and symbolic methods creates a powerful **neuro-symbolic refinement cycle**. In this approach, deep learning models and symbolic learners iteratively improve each other. For instance, one could start with a deep self-supervised model that learns latent features from historical data (capturing complex temporal correlations). These latent features are then distilled into candidate symbolic features (using the method above). Now, treat those symbolic formulas as new features and input them into a simple

model (or directly backtest them) to gauge their standalone efficacy. The feedback might show that some distilled features work well (generalize) while others don't. The ones that work can be retained as **permanent interpretable features**, and the ones that don't can guide the deep model on where it might be overfitting. The deep model can then be retrained or regularized with this knowledge (for example, one can add a penalty if the deep features cannot be approximated by any simple formula, forcing the neural net to learn more "reasonably smooth" patterns). This creates a loop: **deep model proposes → symbolic model distills → evaluate & select → inform deep model retraining**.

An example of neuro-symbolic synergy is using a library like PySR in tandem with deep feature generation: PySR can search the space of formulas to find an **analytical expression that matches a deep feature's behavior**, effectively providing an explicit hypothesis for what the deep network might have found [9]. Another example was mentioned earlier: an RL agent benefiting from symbolic features. We could envision the RL agent itself generating a rough policy (or Q-function) that is then approximated by a symbolic rule; if the symbolic policy performs similarly, it can replace the black-box policy for simplicity and reliability. Research in meta-RL for trading showed that **even a sophisticated meta-learning agent can gain from ingesting symbolic patterns** as features [2] – indicating that symbolic representations provide complementary strengths (e.g. stability, inductive bias) to purely learned representations.

In building the architecture, a *neuro-symbolic feature module* could run in parallel with pure ML feature generation. For example, after each training cycle, take the top K performing deep features or model signals and send them to a symbolic regression process. The outputs (simplified formulas) then go through a performance test (maybe a quick backtest on multiple periods). High-performing ones are added to a **feature library**. The next training cycle, the deep model is augmented with these library features as additional inputs (or it could even be constrained to use them), and the RL or training loop continues. Over time, the system accumulates a set of **validated, interpretable features** distilled from the neural net's wisdom. This not only improves transparency but can also act as a form of regularization on the feature space – since symbolic formulas are inherently less complex than an unrestricted neural net, the model that relies on them is less likely to overfit bizarre patterns.

Retraining Cycles and Pipeline Efficiency

Maintaining a closed-loop system in production requires efficient data pipelines and smart retraining schedules. Financial markets produce vast amounts of data, so the feature generation and backtesting loop needs to be as automated and parallelized as possible. A typical pipeline might operate like this: at regular intervals (e.g. daily or weekly), the latest data is fetched and appended to the dataset; a job is triggered that runs the feature generation module (possibly seeding it with previous best features to refine). This generates new candidate features or updates feature parameters. Next, the system trains a lightweight model (or uses an existing model) to quickly evaluate each feature's predictive power or trading PnL on a validation set. Bad features are filtered out, and promising features move on to a more intensive evaluation with a full backtest. This **staged evaluation pipeline** (quick filter then thorough backtest) saves compute and focuses resources on likely-good features.

When a new feature passes the backtest criteria (e.g. it yields positive returns and decent Sharpe in multiple past scenarios), it can be added to the active model. The model (say a strategy or predictor) is then **retrained** incorporating the new feature. Retraining could be done from scratch on the full history (to avoid biasing too much to recent data), or via incremental learning (updating weights from the last state). The

retraining cycle might also involve **dropping old features** that have degraded in performance (to control complexity and adapt to regime shifts). In effect, the pipeline is continuously *curating* the feature set.

To make this efficient, one can employ parallel processing and asynchronous updates. For example, while one set of features is being backtested, the system can concurrently run the next feature search iteration on another thread or machine. Modern cloud-based pipelines might use event-driven triggers: e.g., when daily data arrives, it triggers the pipeline to update features and models (possibly using something like Apache Airflow or other workflow managers for MLOps). **Caching** is important too – store results of past feature tests to avoid re-evaluating similar features repeatedly. If using an RL agent for feature generation, one can warm-start it with the last policy instead of training from scratch each day, so it gradually continues exploring where it left off.

The retraining cycle frequency should balance freshness with stability. Too frequent retraining (e.g. every minute) might cause thrashing and overfitting to noise; too infrequent (e.g. yearly) might miss important market shifts. Many practitioners opt for a **rolling retrain** (say monthly or quarterly) with interim online updates if needed. Continual learning algorithms as discussed can update more frequently in a stable way. It's also useful to maintain **multiple models** in parallel: one currently live, and one in training with new features – if the new one shows improvement in out-of-sample tests, a handover or blending can occur (this mitigates risk of deploying a bad update).

In summary, an efficient closed-loop pipeline relies on automation (to handle data and retraining without manual intervention), intelligent scheduling, and resource management to continuously evolve features. By structuring the pipeline with clear stages (data -> feature proposal -> quick eval -> full eval -> model update) and using modern MLOps tools, the loop can be maintained with minimal human oversight, while still allowing human-in-the-loop at checkpoints for governance (e.g. approving a new feature if required).

Preventing Overfitting to Backtests

One of the biggest risks in a closed-loop feature discovery is **overfitting to historical backtest data**. The system might find features that look extremely profitable on past data but are actually spurious patterns that won't repeat. Without checks, an aggressive automated loop could over-optimize to noise (a phenomenon sometimes called *backtest overfitting* or *data snooping*). To mitigate this, several strategies must be baked into the process:

- **Use Cross-Validation and Out-of-Sample Tests:** Instead of trusting a feature's performance on a single backtest period, require it to prove itself across multiple *out-of-sample* segments. For example, perform *time-based cross-validation* (walk-forward testing) where a feature is generated on one period and tested on another. A systematic feature discovery study emphasized testing every candidate across multiple time-folds, and treating **performance consistency across folds as equally important as overall performance** [11]. Only features that show stable gains in several distinct periods (e.g. bull and bear markets) are kept. This prevents the loop from seizing on one-time lucky combinations.

- **Multiple Comparisons Control:** The loop might try thousands of features – increasing the chance of false positives. It's crucial to apply statistical techniques to control for this. Methods include Bonferroni or False Discovery Rate adjustments on performance metrics, or requiring a feature's backtest *t-statistic* to exceed a high threshold. In practice, this could mean simulating a large number

of random features to set a baseline for performance (so the system knows what "normal" noise looks like). Features must beat this noise baseline convincingly.

- **Simplicity and Regularization:** Enforce simplicity in feature forms to reduce overfitting. As noted, one way is to evaluate features using **low-complexity models**. For instance, Glassnode's feature exploration used shallow decision trees to test combinations, ensuring that any detected signal was genuine and not an artifact of a very complex model [12]. The idea is that if a simple model (with limited degrees of freedom) can profit from the feature, it's more likely a real pattern rather than overfit. Additionally, regularize the feature search: penalize overly complex features (e.g. very long formulas or those that depend on many parameters) and favor parsimonious ones.

- **Rolling Origin Evaluation:** Continuously monitor how features that were selected are performing on *unseen data going forward*. If a feature was added to the model based on past data, track its live or forward performance. If it sharply underperforms relative to expectations, that's a red flag it might have been an overfit. The pipeline can then drop or down-weight such features. Essentially, the loop should have a **"kill switch"** for features that do not validate in live or pseudo-live tests.

- **Domain Knowledge Filters:** Incorporating human domain knowledge can also help prevent overfitting. Before a new feature is deployed, a human expert or an automated rule could vet whether the feature makes economic sense. Features that are completely non-intuitive might be more suspect (though not always – sometimes AI finds truly novel insights). Nonetheless, having a sanity check like "Does this feature have a plausible rationale or relate to a known market inefficiency?" can add a layer of protection against fool's gold.

By applying these practices, the closed-loop system is forced to focus on **robust signals rather than fleeting coincidences**. It essentially builds in a bias towards generalization. For example, requiring **broad coverage and avoiding bias in the search** (sampling feature space widely rather than just around known good areas) and then filtering out anything that doesn't generalize beyond the training sample is a recipe for finding real patterns [12]. The loop thereby becomes self-correcting: features that were chosen by overfitting tend to fail the next round of out-of-sample tests and get eliminated, preventing them from propagating.

Monitoring Generalization Across Regimes

Financial markets undergo regime changes – periods of bullishness, bearishness, high volatility, low liquidity, etc. A feature might work well in one regime and falter in another. Therefore, the feature learning loop should explicitly **monitor performance across different market regimes** to ensure generalization. This involves both testing in varied conditions and ongoing post-deployment surveillance:

- **Regime-Based Backtesting:** Segment historical data into regimes (which could be defined by volatility level, economic cycle, market stress periods, etc.) and evaluate candidate features in each. For instance, one can check that a feature that claims to predict stock returns works not only in the low-volatility 2010s but also in the high-volatility 2020-2022 period. If a feature's performance comes solely from one regime and is absent or negative in others, it might be too regime-specific. The selection criteria can require that a feature contributes in at least two distinct regimes or include a penalty for high variance in performance across regimes.

- **Rolling Window/Yearly Analysis:** A straightforward approach used in studies is to examine how feature performance **evolves over time**. For example, evaluate the feature's return or Sharpe in a sliding window year by year. One case study found that even top features showed **declining performance trajectories from 2017 to 2023, suggesting increasing market efficiency or changing dynamics eroding their edge** [13]. By plotting such trajectories, the system can catch when a feature's efficacy is decaying as regimes shift. Features whose performance trend is sharply downward can be flagged for replacement or retraining in the loop.

- **Regime Switching Models:** The pipeline can incorporate a regime detection mechanism. For example, a separate model could detect when the market has entered a different state (using metrics like volatility, correlations, macro indicators). The feature generator might then be run separately for each regime to find specialized features, and an ensemble model could **switch or weight features depending on the current regime**. Monitoring generalization then means ensuring the *portfolio of features* covers all regimes. You prevent over-reliance on any single regime's features by maintaining diversity.

- **Out-of-Sample and Live Monitoring:** Even after rigorous historical testing, true generalization is confirmed only with future data. As the model runs live (or on a paper-trading forward test), track its performance in each new regime encountered. If a new type of market condition appears (say a pandemic crash or a sudden structural break), analyze how the features are doing. The closed-loop system should be ready to *learn new features* if none of the existing ones handle the new regime well. Continual learning, as discussed, aids in this by updating with new data. Additionally, maintain a **dashboard of performance metrics** by regime – e.g. the Sharpe in last 1 month vs last 12 months, or performance in high-vol vs low-vol days – to catch any regime-specific degradation quickly.

- **Stress Testing and Scenario Analysis:** As a proactive measure, one can simulate how features would perform under hypothetical or historical extreme scenarios (e.g. 2008 crisis, 1987 crash if data available). This can be part of the feature evaluation step. If a feature dramatically fails in a scenario that wasn't in the training data, that indicates lack of robustness. It might still be included but perhaps with a safeguard (like a regime filter that turns it off in that scenario).

Monitoring generalization across regimes ensures the features maintained by the loop are **truly robust and not just narrow specialists**. It's acceptable to have some regime-specific features (one could explicitly maintain different feature sets for different regimes), but the system should know when each applies. The feedback loop can incorporate regime performance as part of the reward – for example, reward could be a weighted sum of performance across multiple regimes to encourage finding features that are broadly useful. In practice, Glassnode's framework did exactly this by using time-based folds as a proxy for different market conditions, ensuring discovered features "work reliably across different market conditions" [11]. By continuously evaluating and re-evaluating features in this way, the system maintains a high-level *awareness* of when a feature is genuinely predictive vs. when it might be exploiting a temporary market inefficiency that could close.

Using Risk-Adjusted Metrics as Optimization Signals

In financial feature engineering, it is crucial to optimize not just for raw returns but for **risk-adjusted performance**. Incorporating metrics like the Sharpe ratio, Sortino ratio, or maximum drawdown into the

feedback loop guides the AI to find features that yield *smoother, more reliable profits*. There are several ways to include risk considerations in the closed-loop:

- **Reward Functions with Risk Adjustment:** In an RL or evolutionary search context, define the reward as a risk-adjusted metric. For example, instead of reward = total return, use reward = Sharpe ratio (which penalizes volatility) or include a drawdown penalty. This directly tunes the search towards strategies that balance risk and return. Researchers often prefer Sharpe for this reason, and in fact policy-gradient algorithms have been designed to **maximize Sharpe ratio directly as the objective** [4]. Similarly, genetic programming searches have used Sharpe as the fitness function, finding strategies superior to those optimized on simple returns [1]. The closed loop thus inherently values a 1% return with low volatility more than a 1% return with wild swings.

- **Differentiable Risk Metrics in Training:** If using a differentiable approach, one can incorporate the formula for Sharpe or Sortino into the loss function. Sharpe ratio $S = \frac{E[R]}{\sigma(R)}$ is differentiable w.r.t. portfolio weights or model outputs (ignoring non-linearity of stdev which is still smooth). Some studies embed such metrics into end-to-end models – *e.g.* treating maximizing Sharpe as the training goal [4]. Care must be taken to handle non-linearity, but it is doable. Alternatively, use a **proxy risk metric** that is easier to optimize (like variance or CVaR as a differentiable loss component) to achieve a similar effect.

- **Multi-Objective Optimization:** In the feature generation loop, sometimes it helps to optimize multiple criteria: e.g. maximize return **and** minimize drawdown. This can be done by forming a single reward that is a weighted combination (for example, $R - \lambda \times \text{MaxDrawdown}$). The system then searches for features that produce good returns without big equity curve drops. The weights λ can be tuned to reflect the risk appetite. One can also explicitly target metrics like **Sortino ratio** (which penalizes downside volatility). The Glassnode feature exploration noted that their framework could just as easily target *risk-adjusted measures like Sharpe or Sortino, or even drawdown characteristics, as the optimization objective* [14]. By switching the optimization signal to Sharpe vs. raw return, they would discover a different set of optimal features [14]. This highlights that the choice of reward metric in the loop fundamentally shapes what patterns the AI will find [15].

- **Risk Constraints during Search:** Another strategy is to impose hard risk limits during feature evaluation. For example, when backtesting a feature's strategy, require that max drawdown < X% or Sharpe > Y for it to be considered successful. If a feature yields high returns but via very volatile swings, the pipeline can flag that as too risky and either adjust it (maybe by scaling positions) or discard it. This prevents the loop from gravitating to extremely risky "blow-up" strategies that only look good on paper.

- **Sharpe Ratio as a Feedback for Model Complexity:** Risk-adjusted metrics can also act as a guard against overfitting – often an overfit strategy will have high return but also very high variance (thus a mediocre Sharpe). By optimizing Sharpe, the loop inherently prefers more stable patterns, which are more likely to be genuine. In the earlier HFT RL example, the authors reported **higher returns *and improved risk-adjusted metrics*** for their feature-enriched RL agent [3], indicating that their training reward likely balanced risk and return. In practical terms, a feature that only works by taking huge leveraged bets will be penalized if Sharpe is the yardstick.

Incorporating risk-adjusted metrics ensures the feedback loop is aligned with investor preferences and real-world viability. A strategy that maximizes only return might pick up penny-stock like features that triple the money one year and lose it all the next; maximizing Sharpe would likely avoid that by favoring consistency. Moreover, using metrics like drawdown explicitly can lead the AI to discover **hedging features** or risk-management signals (for instance, a feature that triggers an exit before big crashes could improve Sortino or drawdown stats significantly and thus be rewarded). These kinds of risk-focused features are valuable for building more resilient trading systems.

Finally, it's worth noting that the loop can incorporate **risk parity or portfolio optimization steps** as part of feature evaluation. For example, once a set of features (strategies) is generated, one could allocate capital among them using mean-variance optimization. The feedback could then be the portfolio Sharpe. This encourages finding a **diverse set of features** that together improve risk-adjusted return (maybe one feature works in downturns, another in upturns, complementing each other). The closed loop thereby doesn't just find *any* predictive features, but those that improve the **overall portfolio Sharpe or reduce drawdown when added to the existing set** – a much more holistic goal consistent with how portfolio managers think.

Conclusion

Implementing a closed feedback loop for feature learning in financial modeling is a complex but rewarding endeavor. By uniting data engineering, machine learning, and rigorous validation, the system becomes a self-improving "alpha factory." Key components include an architecture that tightly links feature generation with performance evaluation, use of advanced strategies like RL and differentiable backtesting to efficiently search the space, continual learning to adapt to new data, and neuro-symbolic methods to maintain interpretability and robustness. Equally important are the safeguards: systematic cross-validation to curb overfitting, regime-based analyses to ensure durability of signals, and risk-aware objectives to produce realistic trading strategies. When done correctly, this closed-loop can **continuously evolve features** that keep a strategy competitive in changing markets, all while providing transparency and control. The result is an AI-driven process that not only finds promising predictive features but also learns from its mistakes, adapts to new regimes, and aligns with the ultimate goal of sustainable, risk-adjusted financial performance.

Sources: The concepts and strategies discussed are informed by recent research and practice in quantitative finance and machine learning, such as reinforcement learning frameworks for trading [3] [2], differentiable policy optimization for portfolio metrics [4] [6], continual learning approaches for non-stationary time series [8], symbolic regression for interpretable model distillation [9] [10], and systematic feature discovery methodologies that emphasize out-of-sample robustness [11] [14]. These sources underline the importance of each component in the loop and provide evidence of their effectiveness in a financial context.

---

[1] aeaweb.org
https://www.aeaweb.org/conference/2024/program/paper/A7Fhha3r

[2] Neuro-symbolic Meta Reinforcement Learning for Trading
https://www.emergentmind.com/papers/2302.08996

3 Feature Enrichment Imitative Reinforcement Learning for High-Frequency Trading by Fei Han, Qing-Hua Ling, Sheng Lu, Henry Han :: SSRN
https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5375707

4 5 6 7 Decision by Supervised Learning with Deep Ensembles: A Practical Framework for Robust Portfolio Optimization
https://arxiv.org/html/2503.13544v3

8 openreview.net
https://openreview.net/pdf/d70c088a86698db3360b9c81f49525f5ed017c83.pdf

9 GitHub - MilesCranmer/PySR: High-Performance Symbolic Regression in Python and Julia
https://github.com/MilesCranmer/PySR

10 proceedings.neurips.cc
https://proceedings.neurips.cc/paper/2020/file/c9f2f917078bd2db12f23c3b413d9cba-Paper.pdf

11 12 13 14 15 Systematic Feature Discovery for Digital Asset Markets
https://insights.glassnode.com/systematic-feature-discovery-for-digital-assets/