# Continual Learning for Transformer/LLM-Driven Portfolio Optimization Under Non-Stationarity

A constraint-aligned design expressed as one end-to-end spine

Anonymous (draft for internal planning)

Online updates are easy to wire up and hard to own. In a portfolio stack, the expensive failures are the quiet ones: a model refresh that looks fine on recent data but changes behavior under stress, pushes the optimizer into corners, or forces you to debug risk after the fact.

We keep this note concrete: a daily-rebalanced, market-neutral, multi-factor US equities long/short portfolio operating through volatility clustering, correlation spikes, and alpha decay. Drift appears as movement in the mapping $(x_t, z_t) \mapsto \mu_t$ from market state $x_t$ and text context $z_t$ to expected *residual* returns.

Throughout—including the toy validation at the end—$\mu_t$ comes from a **Transformer language model**. Text is tokenized into $z_t$; the Transformer backbone is frozen; and the only parameters we update and ship are a small LoRA adapter.

The interface is intentionally boring. The model may change, but it may only emit an alpha vector $\mu_t$. The decision layer deterministically maps $\mu_t$ to weights $w_t \in \mathcal{W}_t$, and we only ship an update if it passes a fixed stress suite treated as a regression test. Two invariants hold as $\mu_t$ drifts:

- **Feasibility by construction:** every traded $w_t$ is feasible because it is computed by optimizing over $\mathcal{W}_t$.
- **Stress non-regression:** an update is rejected if it worsens performance on a fixed stress suite (historical regimes and tail events) treated as a test fixture.

## 1 Principle: a hard decision boundary

Trading needs a contract, not a training loop. Each day we take a vector of residual alphas $\mu_t$ and turn it into a portfolio by solving a constrained program. The hard limits live in the optimizer, so they do not depend on the model version.

Each day $t$ after close, we compute target weights $w_t \in \mathbb{R}^{N_t}$ by solving

$$\max_{w \in \mathcal{W}_t} \ \mu_t^\top w \ - \ \tfrac{\gamma}{2} w^\top \Sigma_t w \ - \ c_t^\top |w - w_{t-1}|. \tag{1}$$

The important choice is where safety lives: hard limits are encoded in $\mathcal{W}_t$, not learned.

A minimal production-style feasible set for this spine is

$$\mathcal{W}_t := \Big\{ w \in \mathbb{R}^{N_t} : \ \mathbf{1}^\top w = 0, \ \|w\|_1 \le L, \ |w_i| \le w_{\max} \, \forall i, \ \|w - w_{t-1}\|_1 \le \tau, \ \|B_t^\top w\|_\infty \le b_{\text{factor}} \Big\}, \tag{2}$$

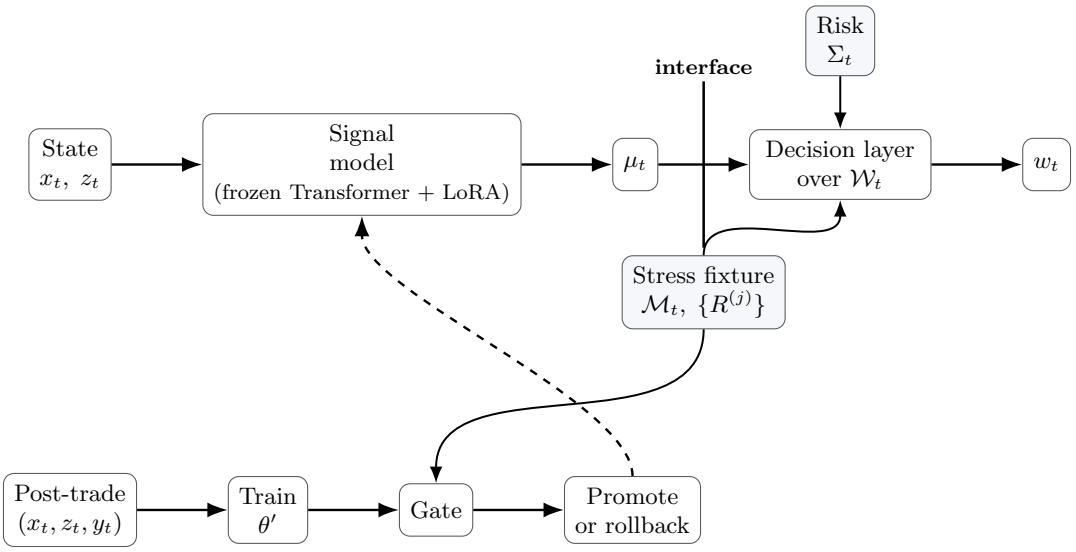


**Figure 1: One interface, two planes.** Trading consumes only $\mu_t$ and enforces feasibility by optimizing over $\mathcal{W}_t$ with fixed primitives $(\Sigma_t, \{R^{(j)}\})$. Updates are treated like releases: we only promote an adapter if it does not regress on the same audited stress fixture $\mathcal{M}_t$.

augmented with tail constraints on a fixed scenario library. For stress windows indexed by $j \in \{1, \dots, J\}$ with scenario matrices $R^{(j)}$,

$$\mathrm{ES}_\alpha(R^{(j)} w) \geq -L_j \qquad \forall j. \tag{3}$$

Problem (1) consumes a learned $\mu_t$ and an auditable risk primitive $\Sigma_t$. In practice, correlations often move faster than alpha, so we treat $\Sigma_t$ as infrastructure and build it from an explicit factor model:

$$r_\tau = B_t f_\tau + \varepsilon_\tau, \qquad \tau \in [t-H, t-1], \tag{4}$$

$$\Sigma_t = B_t\, F_t\, B_t^\top + D_t, \qquad F_t := \mathrm{Cov}_t(f), \quad D_t := \mathrm{diag}(\mathrm{Var}_t(\varepsilon_i)). \tag{5}$$

If you collapse the boundary, failures stop being legible. Letting the model emit weights turns feasibility into after-the-fact monitoring. Training on raw returns quietly reintroduces factor bets through the back door. Updating large shared parameter blocks without a fixed stress fixture makes rollback expensive and guarantees you will forget rare regimes at exactly the wrong time.

## 2 Mechanism: constrained continual updates

We update only what we can safely roll back: a small LoRA adapter. Two choices keep the update loop aligned with the trading loop: (i) train on residual returns so the model does not relearn factor premia, and (ii) treat the stress suite as a regression test that the update step is not allowed to break.

Residual supervision keeps the learning target stable:

$$y_t := r_{t+1} - B_t f_{t+1} \in \mathbb{R}^{N_t}. \tag{6}$$

The signal stack predicts expected residual returns $\mu_t = \mu_\theta(x_t, z_t)$. The Transformer text backbone is frozen; the only thing we update online is a small LoRA adapter in the head. On the current slice, a simple supervised loss is enough:

$$\ell_t(\theta) = \sum_{i=1}^{N_t} \big(y_{i,t} - \mu_{i,t}\big)^2. \tag{7}$$

Here, "memory" is not an abstract task list. It is a fixed test fixture: a curated set of recent points plus stress regimes and edge cases,

$$\mathcal{M}_t := \mathcal{M}_t^{\text{recent}} \;\cup\; \bigcup_{j=1}^{J} \mathcal{M}^{\text{stress}(j)} \;\cup\; \mathcal{M}_t^{\text{edge}}. \tag{8}$$

The alignment is structural: the same stress windows define both (a) the replay set we refuse to forget and (b) the scenario library $R^{(j)}$ used in (3).

Each update is allowed to fit the current slice, but it is not allowed to move freely on $\mathcal{M}_t$. Two stabilizers do the work:

- **Replay distillation (behavior anchor).** Penalize movement away from the deployed model on replay contexts:
$$\ell_t(\theta) + \lambda\, \mathbb{E}_{(x,z)\sim\mathcal{M}_t}\big[\big(\mu_{\theta_t}(x,z) - \mu_\theta(x,z)\big)^2\big]. \tag{9}$$
- **Gradient projection (hard non-regression).** For stress blocks $\mathcal{D}^{(j)} \subset \mathcal{M}_t$ with gradients $g_j = \nabla_\theta \ell_{\mathcal{D}^{(j)}}(\theta)$, project the current gradient $g_{\text{cur}}$ to satisfy
$$\min_v \;\tfrac{1}{2}\|v - g_{\text{cur}}\|^2 \quad \text{s.t.} \quad \langle v, g_j\rangle \geq -\epsilon_j \;\; \forall j. \tag{10}$$

A candidate adapter is promoted only if it passes the fixed gate suite on $\mathcal{M}_t$; rollback is a single adapter swap.

## 3 Evidence: minimal executable validation

This experiment is intentionally small and opinionated: it is a regression test for the contract above, not a benchmark. The point is to show the failure mode ("tracks drift, forgets stress") and the fix ("adapts without breaking the stress fixture") end to end.

The synthetic market (in `example_code/`) simulates $N$ assets with a low-rank factor structure, regime-switching factor covariance, and a regime-dependent alpha mapping, with two forced crisis windows used to define the stress fixture. The testbed is LLM-based: the simulator also emits a token stream whose distribution shifts with regimes; a tiny causal Transformer LM is pretrained once and frozen to produce text embeddings $z_t$.

We compare two update policies with the same frozen Transformer backbone, the same decision layer, and the same initialization: **naive** (current-slice SGD) vs **cl** (replay distillation + projected updates + a promotion gate with rollback). We call it a success if stress loss does not regress, IC stays responsive, and the ES budget remains satisfied by construction.

*Reproducibility:* `python example_code/run_validation_experiment.py` regenerates the embedded figures.

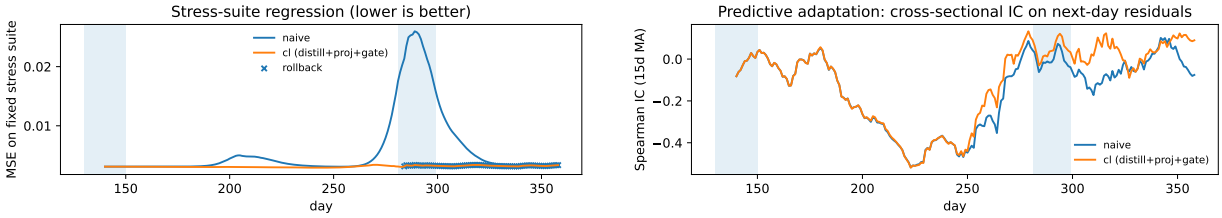


**Figure 2: Core update-loop check. Left:** MSE on a fixed stress fixture (a regression test). Naive online updates eventually forget: stress loss spikes after later drift. Constrained updates stay pinned; × marks are rejected releases (rollback). **Right:** 15-day MA of cross-sectional Spearman IC on next-day residuals. The constrained updater stays responsive while protecting the stress fixture.

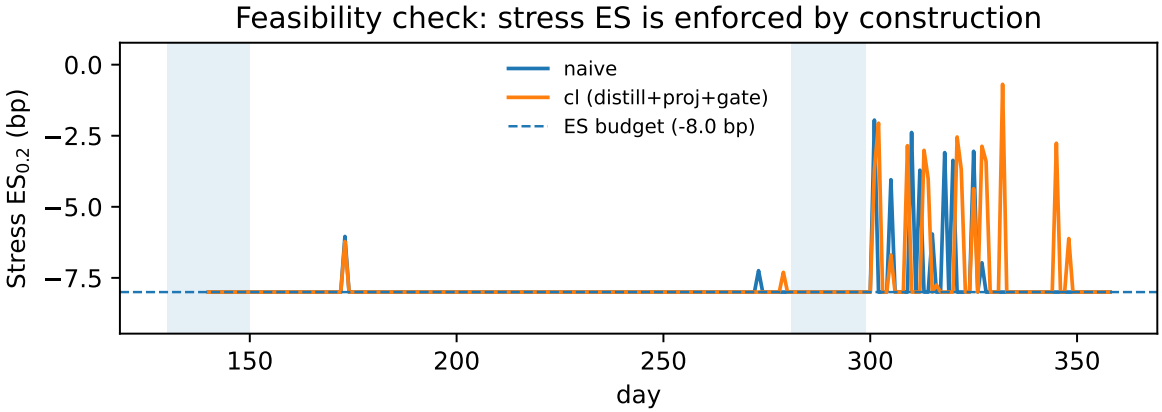


**Figure 3: Feasibility is not a learning problem.** Stress Expected Shortfall on the fixed scenario library (in bp) plotted against the decision-time budget. The trace sits on the budget because the decision layer scales exposure until (3) is satisfied; the updater can change $\mu_t$ but it cannot violate $\mathcal{W}_t$.

Taken together: Figure 2 shows why the updater must be constrained (stability does not emerge on its own), while Figure 3 shows why feasibility belongs in the optimizer (it stays invariant under model drift).