Optimal Execution with Reinforcement Learning by Team 4

Team 4

July 17, 2025

Why Optimal Execution Matters

- Large orders move prices *(market impact)* poor execution timing can significantly increase costs.
- Execution quality is measured by metrics like Implementation Shortfall (IS) and Expected Shortfall (ES).
- Classic solution: AlmgrenChriss (AC) model provides an optimal trading schedule under idealized assumptions.
- But real markets have richer microstructure: non-linear impact, varying liquidity, fees, etc., challenging the AC model.

The Almgren–Chriss Model

- Optimal execution cast as a meanvariance trade-off: minimize expected trading cost + $\lambda\times$ (variance of cost).
- Assumes simple price dynamics (Arithmetic Brownian Motion with zero drift) and linear market impact.
- Temporary impact: each trade incurs a linear price penalty (slippage); Permanent impact: each share traded permanently shifts the price linearly.
- Closed-form solution: a static optimal sell schedule parameterized by risk aversion λ. Varying λ traces an efficient frontier of cost vs. risk.

Limitations of the AC Model

- Assumes price follows a simple arithmetic random walk (no jumps, no changing volatility) or a fixed drift not true in real markets.
- Impact is linear with volume; actual market impact can be non-linear (e.g. higher cost per share for large block trades) and depends on market depth.
- Only captures risk via variance of cost; ignores extreme tail-risk and other risk metrics that traders care about.
- Yields a static trading schedule cannot adapt or react to intraday price movement or news once execution begins.

Why Reinforcement Learning?

- **Model-free and adaptive:** No need to assume a specific price model an RL agent can learn the dynamics from data and adjust actions based on real-time price observations.
- Naturally handles high-dimensional state inputs (price history, market signals) and continuous action outputs (order sizes).
- Can optimize non-linear and custom objectives beyond mean-variance (e.g. directly minimize tail-risk or incorporate transaction costs).
- Enables **dynamic strategies**: policy updates its decisions in reaction to market movements during execution (unlike ACs static plan).

State Representation

State vector $s_k = [\, r_{k-D+1}, \ldots, r_k,\ m_k,\ i_k \,]$

- $r_{k-D+1:\,k}$ = window of last D log-returns
  - captures recent momentum and volatility in a scale-free manner
  - $D = 5$ is a balance: enough history for trends, but keeps dimensionality low
- $m_k = \frac{T-k}{T}$ = fraction of time remaining
  - acts as an urgency clock: near end, execution must accelerate regardless of price
- $i_k = \frac{Q_k}{Q_0}$ = remaining inventory fraction
  - indicates progress: whether most shares remain or execution is nearly complete

Tuning D**:**

- Shrink ($D < 5$): faster training, less overfitting, but may miss longer momentum
- Expand ($D > 5$): captures longer autocorrelation, but increases state size and training cost
- Treat D as a hyperparameter: grid-search over values (e.g. $\{3, 5, 8, 12\}$) and select via execution-cost vs. risk trade-offs

Action Space

- Continuous action $a_k \in [0, 1]$: the fraction of current remaining inventory to sell at time k.
- Executed shares $Q_{\text{sell},k} = a_k \times Q_k$ (where Q_k is shares left at step k).
- Boundaries: $a_k = 0$ means do not trade at this step; $a_k = 1$ means sell *all* remaining shares immediately.
- This fractional approach naturally prevents overselling and short selling. (Agent cannot sell more than whats left, and cannot go negative.)
- *We will also experiment with alternative action interpretations (e.g. treating a_k as a target trading rate or scaling it by market conditions), but the base case is simple proportional execution.*

Reward Functions in RL Execution

- **PnL (Profit and Loss)**: Measures change in portfolio value: $\Delta(\text{Cash} + \text{Inventory} \times \text{Price})$
- **Running Inventory Penalty**: Adds quadratic penalties for holding inventory across time and at terminal step.
- **CjMm Criterion**: PnL − running inventory aversion − terminal inventory mismatch.
- **CjOe Criterion**: PnL − per-step inventory aversion − exponentially weighted leftover impact.
- **Exponential Utility**: $\mathbb{E}[-\exp(-\gamma(X_T + Q_T S_T))]$, introduces strong risk aversion.
- **Normalized Execution Reward**: $(r_t - Q_t P_0)/Q_{\text{total}}$ normalized version of execution revenue.
- **Execution Shortfall + Penalties**:
  - Penalizes shortfall: $Q_t(P_0 - P_t)$
  - Adds: temporary market impact, leftover penalty, trade smoothness, inventory fraction explosion.

DDPG: Baseline ActorCritic

- **Deep Deterministic Policy Gradient** (Lillicrap *et al.*, 2015)
- Off-policy actorcritic for continuous actions
- *Actor* $\pi_\phi(s)$ outputs deterministic action $a \in [-1, 1]$
- *Critic* $Q_\theta(s, a)$ learns value via TD error
- Exploration via OrnsteinUhlenbeck (or Gaussian) noise
- **Pros:** Sample-efficient, reuses replay buffer
- **Cons:**
  - Over-estimates Q (no double-Q)
  - Unstable when reward / dynamics are noisy
  - Sensitive to hyperparameters

Deep Deterministic Policy Gradient (DDPG)

- Off-policy, actorcritic RL algorithm designed for continuous action spaces.
- **Actor:** a neural network $\pi_\theta(s)$ outputs a deterministic action a given state s. **Critic:** a network $Q_\phi(s, a)$ estimates the Q-value (expected return) of state-action pairs.
- Learns via alternating updates: Critic learns by minimizing TD error; Actor learns by gradient ascent on Q (improve actions w.r.t. critics evaluation).
- Uses experience replay (random mini-batches from past experiences) and target networks for stability. Exploration is driven by adding noise (e.g. OrnsteinUhlenbeck process) to the actors actions during training.

Dense vs. Sparse Rewards

- **Dense:** Agent receives reward feedback at each time step (e.g. based on the immediate trading cost or P&L change after each action).
- **Sparse:** Agent receives zero reward during the episode and only gets a final reward at the end (based on total execution cost or utility).

Dense vs. Sparse Rewards

- **Dense:** Agent receives reward feedback at each time step (e.g. based on the immediate trading cost or P&L change after each action).
- **Sparse:** Agent receives zero reward during the episode and only gets a final reward at the end (based on total execution cost or utility).

Observation: Dense rewards tended to accelerate learning (more frequent feedback), whereas sparse rewards encouraged more risk-aware behavior but required longer training to converge.

Sparse Rewards

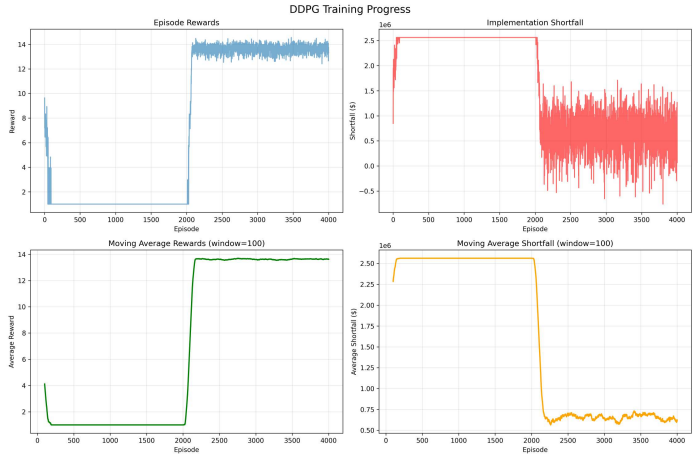


Figure: ddpg baseline training progress

Sparse Rewards

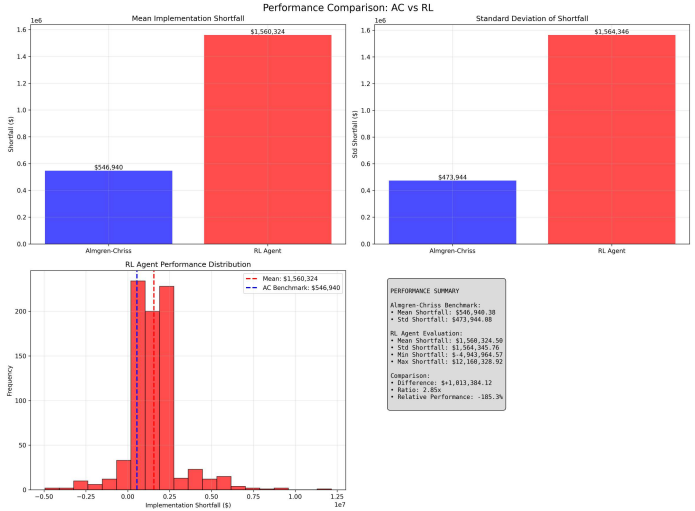


Figure: comparison with the given model

Sparse Rewards

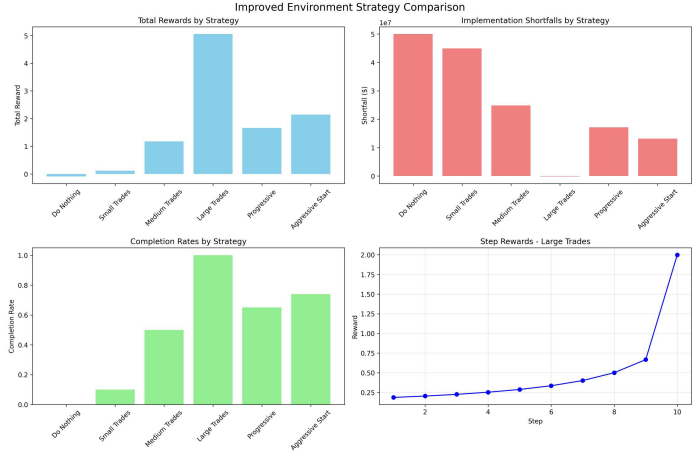
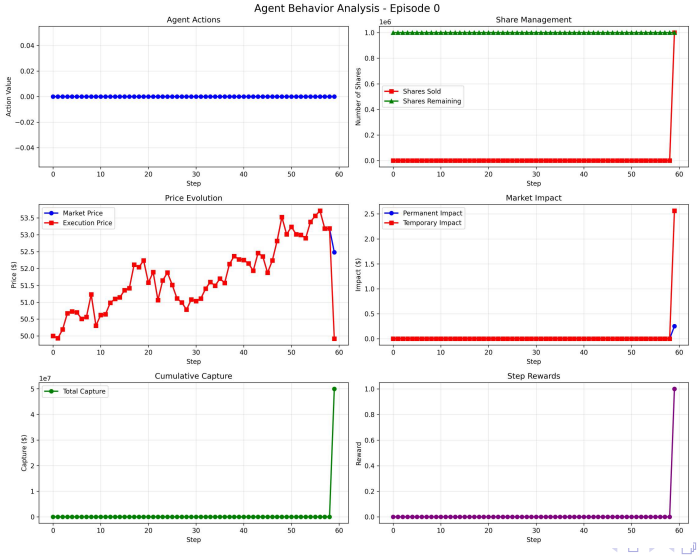


Figure: enhanced environment

Sparse Rewards



Dense Rewards

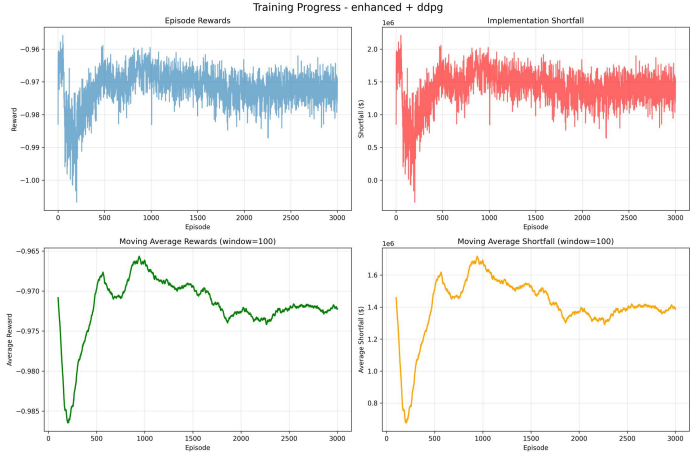
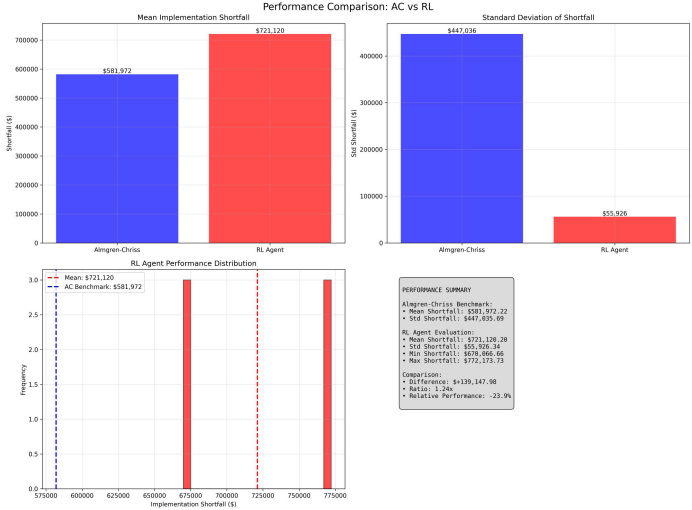


Figure: ddpg dense reward training progress

Dense Rewards vs Almgren Chriss Model



Figure

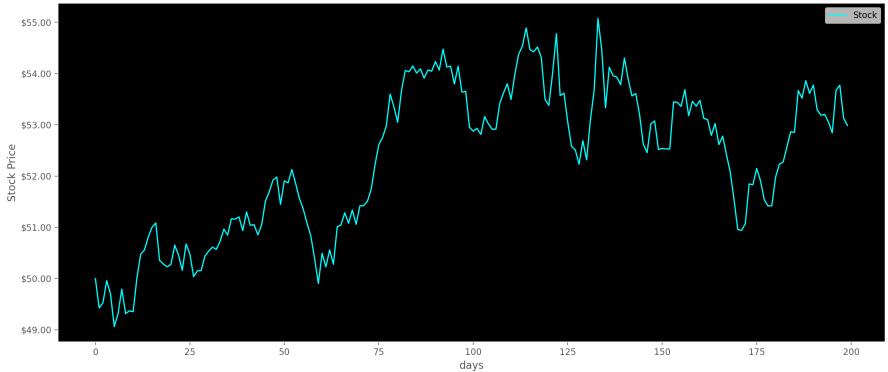
Beyond ABM: GBM and AR(L) Price Dynamics

- **Arithmetic Brownian Motion (ABM):** ACs baseline model price changes ΔS are normal with constant variance, no memory. (Can lead to unrealistic behavior like negative prices over long horizons.)
- **Geometric Brownian Motion (GBM):** More realistic model: $dS_t = \mu S_t\, dt + \sigma S_t\, dW_t$. Prices evolve multiplicatively, so volatility is proportional to price and prices stay positive.
- **Autoregressive returns (AR(L)):** Instead of i.i.d. returns, assume $r_t = \phi_1 r_{t-1} + \cdots + \phi_L r_{t-L} + \varepsilon_t$. Introduces serial correlation (e.g. momentum or mean-reversion) in price movements.
- Real markets exhibit features like short-term momentum and volatility clustering. AR models capture the former, and can be combined with volatility models (e.g. GARCH) to capture time-varying volatility.

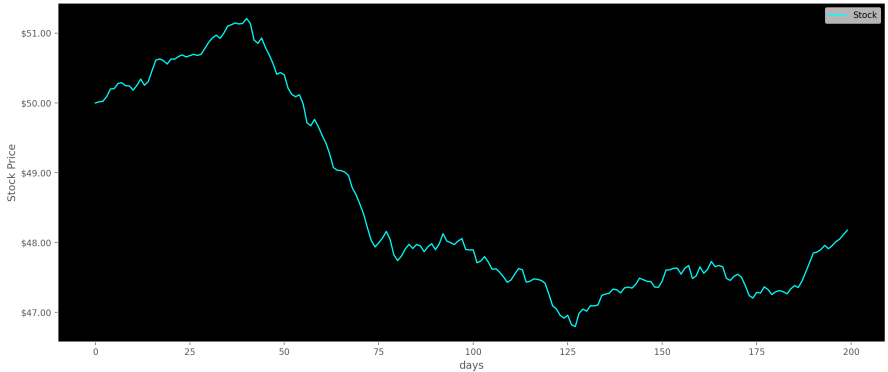
Impact of Changing Price Dynamics

- **Higher volatility (GBM vsABM):** When price volatility is proportional to price (GBM), large swings become more likely. This increases tailrisk of not finishing in time at a good price our CVaRfocused agent responds by executing faster (reducing exposure), especially after volatility spikes. ACs strategy, being fixed, cannot adjust to volatility on the fly.
- **Downward drift:** In a falling market ($\mu < 0$), the optimal play is to sell early. The RL agent learns to *frontload* trades; ACs analytic schedule would do the same only if it already knew the drift.
- **Upward drift / momentum:** With persistent positive returns (AR(L) momentum) the RL agent sells more slowly to capture the upmove, whereas ACs static plan cant time this.

Impact of Changing Price Dynamics



Impact of Changing Price Dynamics



Trading Costs and Market Impact Enhancements

- **Fixed fees:** We include a per-share cost ϵ for trading (e.g. commissions or bid-ask spread). ACs model had a fixed cost component; we carry that over so each share sold incurs a small fixed cost.
- **Non-linear impact (quadratic):** In addition to linear impact, we add a term proportional to v^2 (square of volume traded in a single step). This models the fact that very large trades eat deeper into liquidity and incur disproportionately higher costs.
- **Effect on strategy:** The quadratic penalty strongly discourages large, instant trades. The agent (and even an optimized AC-like strategy) will favor spreading out the order more evenly to avoid the extra cost of big trades. Execution becomes smoother and less impulsive.

DDPG Baseline Performance

- Trained a DDPG agent in the simulated environment (episodes represent full order execution). Used replay buffer of size 10^4, batch size 128, $\gamma = 0.99$, and OU noise for exploration.
- Convergence: the agents policy stabilized after sufficient training, learning to balance impact cost and risk. (Neural network: 2 hidden layers of 24 and 48 neurons in actor/critic.)
- Outcome vs. AC: The DDPG agent with our custom reward achieved a significantly better 95% Expected Shortfall than the AC baseline (i.e., reduced tail-cost). *Preliminary result:* $\text{ES}_{0.95}$ improved by over ACs strategy for comparable average cost.

Performance Across Market Scenarios

- **Simple ABM (random walk, no drift):** DDPG agents strategy looks similar to ACs optimal schedule. With no predictable trends, theres little room to improve on AC both achieve near-optimal cost. (RL essentially recovers the AC solution in this regime.)
- **Trending market (GBM with drift):** In a market with drift, AC (if unaware of drift or unable to adapt) performs suboptimally. Our RL agent adapts: for $\mu < 0$ (downtrend) it sells faster than AC, for $\mu > 0$ (uptrend) it sells more slowly. This adaptability lets RL beat AC in terms of final execution cost.
- **Autocorrelated returns (AR(L)):** ACs static strategy cannot exploit short-term autocorrelation (e.g. momentum bursts or mean-reversion). The RL agent does leverage it e.g., delaying sells during a momentum upswing or holding fire during a dip expecting a rebound thus achieving lower cost/risk.
- Overall, the more the price process deviates from ACs assumptions (pure random walk), the larger the performance gap where RL outperforms AC.

Experiments: Action Variations

- **Baseline Action:** Sell a constant proportion of shares remaining:

$$a_k \cdot \texttt{constant\_shares\_to\_sell}$$

- **Spread Adjusted:** Modify a_k based on bidask spread and recent price momentum:

$$a_k \cdot (1 + \texttt{spread\_factor} \cdot \texttt{momentum})$$

- **Volatility Adjusted:** Scale a_k inversely with volatility to reduce risk during high-variance periods:

$$a_k \cdot \frac{\texttt{volatility}}{\texttt{daily\_price\_variation}}$$

- **Time Weighted:** Increase execution as time runs out:

$$a_k \cdot \frac{T - t_k}{T}$$

- **Volume Constrained:** Cap execution based on a fraction of average daily volume.
- **Percent of Total Shares:** Execute a fixed fraction of original shares regardless of state.
- **Rate Based:** Sell shares proportional to trading speed: $a_k \cdot \frac{Q_k}{T}$

Experiments: Reward Variations & Impact

- **Reward Tuning Strategies:**
  - Compared pure PnL, exponential utility, and inventory-penalizing rewards.
  - Evaluated dense stepwise rewards vs. terminal-only objectives (e.g., exponential utility, CjMm).
  - Used penalty terms for leftover inventory, large trades, and burstiness.
- **Behavioral Effects:**
  - Terminal-penalizing rewards led to earlier liquidation and smoother execution profiles.
  - Exponential utility and CjOe reduced aggressive trading under volatility, improving stability.
  - Inventory penalties encouraged consistent trading over time and reduced end-of-horizon pressure.
  - Normalized rewards led to consistent behavior across assets with varying price scales.
- **Best Configuration:**
  - **CjOe + volatility-aware action mapping:** balanced cost minimization and execution consistency under noisy price dynamics.

Optimal DDPG Average Shortfall

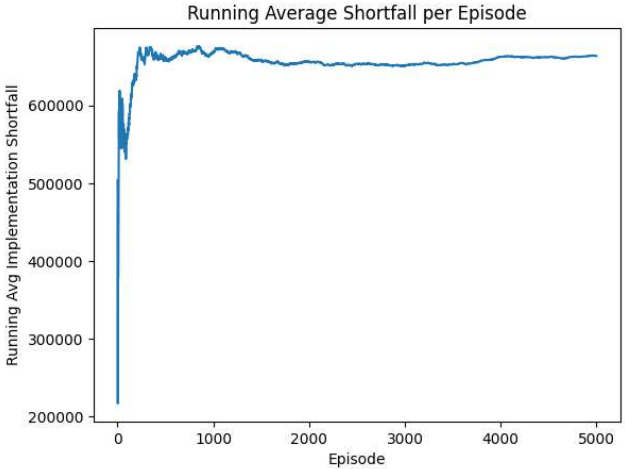


Figure: Average implementation shortfall of DDPG agent training in GBM environment

Optimal DDPG Reward Per Episode

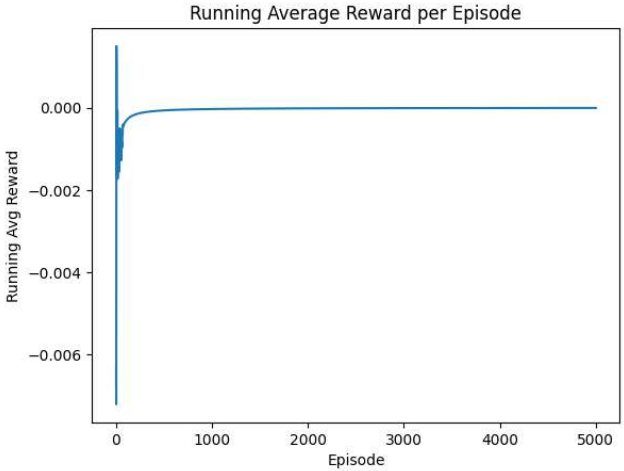


Figure: Reward of DDPG agent training in GBM environment

Best DDPG Result

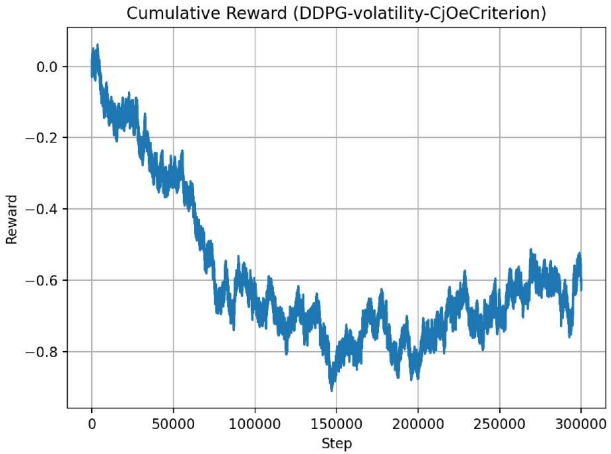


Figure: cumulative reward of training on CjOe + volatility-aware action

DDPG test against Almgren Chriss Model

Method	Mean ($)	Std. Dev. ($)
Agent	604341.86	406969.01
AC	1735125.39	2391435.81

Limitations of DDPG in Execution

- **Q-value overestimation:** DDPGs single critic can develop an optimistic bias, overestimating the value of certain actions, which may lead the policy astray (taking on more risk than intended).
- **Training instability:** Delicate hyperparameter tuning is needed. We observed that slight changes (learning rates, noise scale, etc.) sometimes caused divergence or very slow convergence.
- **Exploration challenges:** With a deterministic policy, DDPG relies on external noise for exploration. If noise is not well-tuned, the agent might not explore some strategies (risk of getting stuck in a local optimum strategy thats suboptimal globally).
- **Sample inefficiency:** DDPG can require a lot of training episodes to reliably learn a good policy (especially with a sparse reward). This could be an issue if using limited real-market data for learning.

TD3: Twin-Delayed DDPG

- **Twin Critic Networks** two Qs to min over, cuts bias
- **Target Policy Smoothing** add small noise to target action
- **Delayed Actor Updates** update policy every d critic steps
- **Pros vs. DDPG:**
  - Far more stable learning
  - Dramatically reduces over-optimistic value estimates
- **Trade-off:** Slightly more compute (two critics), but same replay buffer

Twin Delayed DDPG (TD3)

- **Twin Critics:** TD3 uses two critic networks Q_1, Q_2 and takes the minimum Q value for actor updates and target calculations. This double Q-learning approach reduces overestimation bias.
- **Delayed Policy Updates:** The actor (policy) network in TD3 is updated less frequently (e.g. every 2 or 3 critic updates). This ensures the critic values are more stable and reliable when the actor learns, preventing unstable oscillations.
- **Target Policy Smoothing:** When computing the target Q values for the critic, TD3 adds a small random noise to the target action (clipped to avoid extremes). This makes the critic target more robust, preventing the policy from exploiting sharp local peaks in Q.
- **Effect:** TD3 provides much more stable training than vanilla DDPG. In our context, it means the agents learning curve is smoother and less prone to sudden divergence. It achieves equal or better execution performance, but with more reliable convergence.

Soft ActorCritic (SAC)

- **Maximum-Entropy Objective:**

$$J(\pi) = \sum_t \gamma^t \Big[r(s_t, a_t) + \alpha \, \mathcal{H}\big(\pi(\cdot|s_t)\big) \Big]$$

- Learns a *stochastic* Gaussian policy $\pi_\phi(a|s)$
- Twin critics + entropy bonus robust value estimates
- Auto-tunes temperature α to achieve desired entropy
- **Pros:**
  - Built-in exploration bonus
  - Very stable under stochastic rewards
  - State-of-the-art in many continuous tasks
- **Cons:** More networks & hyperparameters (but auto-tuning helps)

Soft Actor-Critic (SAC)

- **Stochastic policy:** SACs actor outputs a probability distribution (e.g. Gaussian) over actions, and actions are sampled from it. This inherent randomness encourages exploration and learning a distribution of good actions.
- **Maximum entropy objective:** SAC doesnt just maximize expected reward it also maximizes entropy (randomness) of the policy. It aims to find a policy that earns high reward *while being as random as possible*. This leads to more robust, exploratory behavior.
- **Twin critics & auto temperature:** Like TD3, SAC uses two critic networks to reduce bias. It also introduces a temperature parameter α to weight the entropy term; SAC can automatically tune α to balance exploration vs. exploitation.
- **Benefits:** In our problem, SACs entropy-driven approach means the agent learns not just a single deterministic strategy, but a distribution of strategies (useful under uncertainty). Training is very stable. SAC can achieve comparable or better performance than TD3, with the bonus of a naturally stochastic policy (which can be advantageous to avoid predictable execution patterns).

Implementing TD3 and SAC

- **Unified Framework:** Both TD3 and SAC were built on the same infrastructure as DDPG.
  - Identical state space, action space, and architecture.
  - Enables direct comparisons under the same environments and reward functions.
- **Shared Hyperparameters:**
  - Same learning rates, batch sizes, replay buffer settings, and episode lengths.
  - Maintains fairness across algorithmic comparisons.

TD3: Twin Delayed DDPG

- **Core Modifications over DDPG:**
  - Two critic networks for bias reduction.
  - Actor updates delayed: once every 2 critic updates.
  - Target policy smoothing: Gaussian noise added to target actions during critic target computation.
- **Outcomes:**
  - More stable training curves than DDPG.
  - Slight improvement in both shortfall and variance.

Best TD3 Result

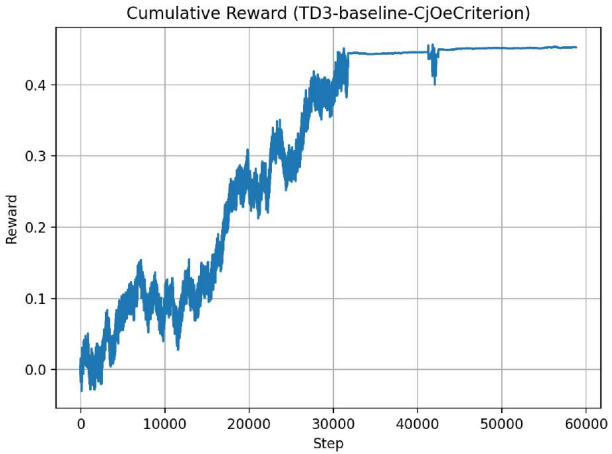


Figure: TD3 training results

Best TD3 Result

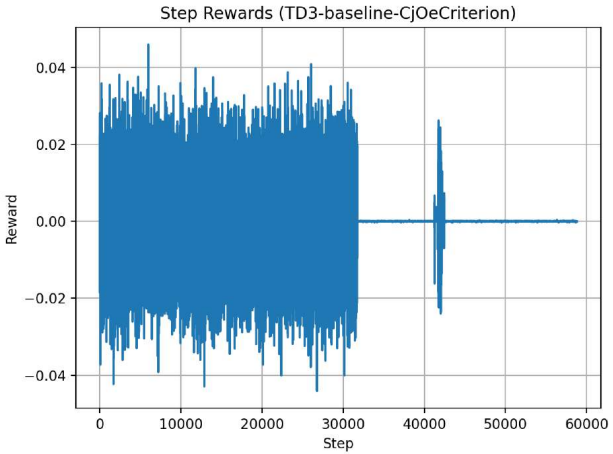


Figure: TD3 training results

Best TD3 Result

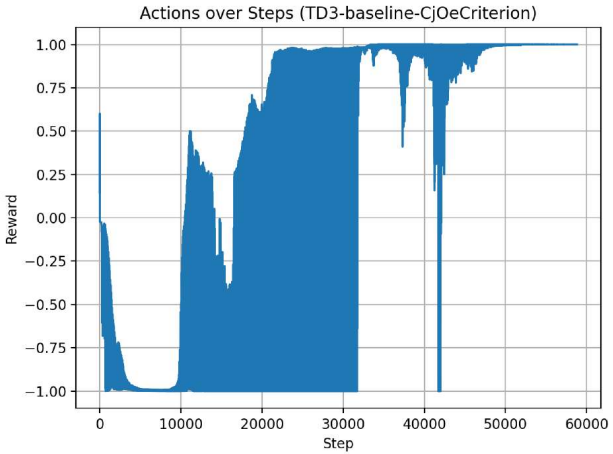


Figure: TD3 training results

TD3 Training Result

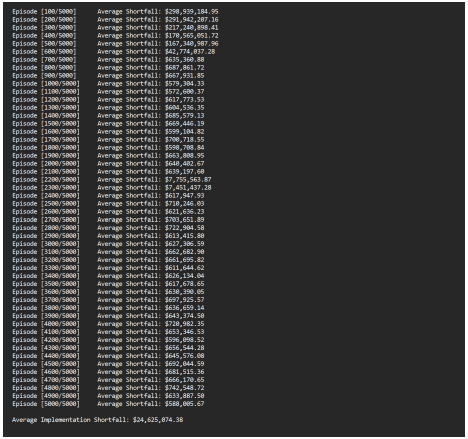


Figure: TD3 training average implementation shortfall

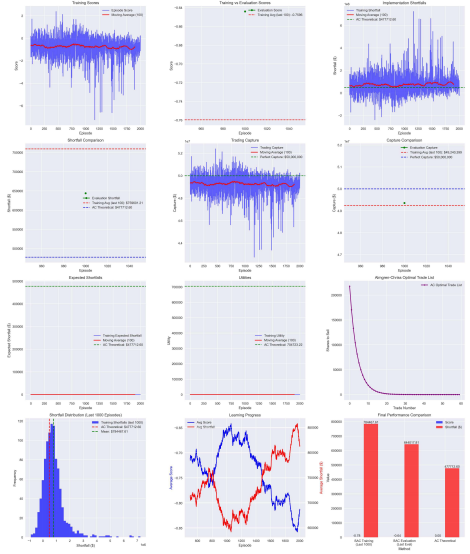
TD3 test against Almgren Chriss Model

Method	Mean ($)	Std. Dev. ($)
Agent	666 096.83	410 591.73
AC	1 732 920.49	2 359 937.15

SAC: Soft Actor-Critic

- **Key Features:**
  - Stochastic actor: samples from learned Gaussian policy.
  - Reparameterization trick with tanh squashing to map actions into $[-1, 1]$.
  - Dual critics for stability; entropy term promotes exploration.
  - Temperature α is tuned to balance exploration and exploitation.
- **Performance:**
  - Most stable policy training.
  - Consistently lowest tail-risk across tested reward functions.

Best SAC Result



AC vs. DDPG vs. TD3 vs. SAC

- **AC (AlmgrenChriss):**
  - Analytically elegant baseline under ABM and linear impact.
  - Inflexible in non-stationary or complex regimes.
- **DDPG:**
  - Learns adaptive, nonlinear execution policy.
  - Strong performance in GBM/AR(L) scenarios.
  - Training is sensitive to initialization and hyperparameters.
- **TD3:**
  - Fixes overestimation bias in DDPG.
  - More consistent and stable learning.
- **SAC:**
  - Stochastic policy improves robustness and tail-risk.
  - Less sensitive to tuning, handles uncertainty better.

Final Recommendation

- **AC is useful as a benchmark**, especially for gaining intuition and computing quick baselines.
- **RL methods dominate in complex environments**: GBM, AR(L), or nonlinear objectives.
- **Best Performer:** SAC provided the most stable and robust execution with lowest tail-risk.
- **Close Second:** TD3 performed slightly better than DDPG, especially in terms of stability.
- **Deployment Suggestion:** Use AC as a sanity check; use SAC as the main model for real-time policy generation.

References I


R. Almgren and N. Chriss. *Optimal Execution of Portfolio Transactions*. 2001.


J. Gatheral and A. Schied. *Optimal Trade Execution under GBM*. 2012.


P. Cheridito and M. Weiss. *Reinforcement Learning for Trade Execution with Market Impact*. arXiv:2507.06345, 2025.


Y. Hafsi and E. Vittori. *Optimal Execution with Reinforcement Learning*. arXiv:2411.06389, 2024.