Problem Setting: In executing a large trade, if we dump shares too quickly, we push the price down; too slowly and we risk adverse price moves. This is the essence of *optimal execution*: how to split a large order over time to minimize cost and risk. Traders traditionally evaluate execution cost via **Implementation Shortfall (IS)**, which is the difference between the trade's actual cost and a benchmark (like the arrival price). They also care about risk measures like **Expected Shortfall (ES)**, which quantifies tail risk (the average of worst outcomes).

Classic Approach: The go-to analytical model is the **Almgren–Chriss (AC)** framework (2001), which gives a closed-form optimal execution plan. AC was groundbreaking because it explicitly modeled *market impact* (the price change caused by trading). However, AC's solution comes with strong assumptions. It assumes a very simple model of price dynamics and impact. In practice, **market microstructure** is far more complex: price movements can be volatile with jumps, liquidity (how easily you can trade) varies over time, and execution incurs fixed fees or non-linear costs. These real-world factors aren't fully captured by AC's basic assumptions, which is why we need to consider more advanced approaches.

AC Model Basics: The Almgren–Chriss model formulates executing an order as an optimization problem. We have a fixed time horizon (say, end of day) to sell a certain quantity Q_0 shares. The goal is to minimize the total cost of selling (including price impact) while managing the risk of price uncertainty. AC achieves this by a **mean-variance optimization**: minimize the expected cost plus a penalty on the variance of cost. Here λ is a risk aversion parameter — if $\lambda = 0$, the trader is risk-neutral and only cares about minimizing expected cost; if λ is large, the trader is very risk-averse and will accept a higher expected cost to reduce variability.

Assumptions: AC assumes the stock price follows an *Arithmetic Brownian Motion (ABM)* (basically, a random walk with constant volatility and possibly a constant drift). Under this model, price changes are independent and normally distributed (no volatility clustering or jumps). Market impact is modeled in two linear terms: - A **temporary impact**: if you sell a chunk of shares, you push the execution price down by an amount proportional to the volume (this impact dissipates by the next time step). - A **permanent impact**: each share sold depresses the price floor a bit permanently (small linear decrease in the fundamental price level).

With these assumptions, AC can derive a **closed-form optimal trading trajectory**. Essentially, it tells you how many shares to sell in each sub-interval of your horizon. The exact solution is static and deterministic (computed at time 0 and not changed thereafter). By tuning λ, you get dif-

ferent strategies: low λ (risk-neutral) might front-load the selling to finish quickly (minimize impact cost at the expense of high risk), whereas high λ spreads out trades more evenly to reduce risk (but incurs more impact cost). All such strategies lie on an *efficient frontier*: you can't reduce cost without increasing risk and vice versa. AC's formula gave traders a clear schedule and insight into the cost-risk trade-off under its assumptions.

While AC is elegant, its simplifying assumptions limit its real-world applicability: - **Simple Price Dynamics:** AC assumes an arithmetic Brownian motion (basically, prices performing a random walk with constant volatility). In reality, asset prices often exhibit volatility clustering, jumps, and other features that break this assumption. If the actual price process deviates (say there's a sudden trend or volatility spike), a schedule precomputed under the random walk assumption may no longer be optimal. - **Linear Impact:** AC's linear impact model is a rough approximation. In practice, if you try to execute a very large volume in a short time, the impact isn't linear – it can worsen disproportionately as you consume more liquidity from the order book (the price impact curve is often concave or has nonlinear components). Also, impact can depend on things like order book shape and other traders' behavior, which aren't captured in a simple linear model. - **Risk Measure:** AC uses variance of execution cost as the risk metric (implicitly assuming a normal distribution of outcomes). But traders might be more concerned with tail risks (how bad the worst losses can get) rather than just variance. For instance, two strategies could have the same variance, but one might have a small chance of a very large loss — variance alone wouldn't distinguish that. - **Static Schedule:** Perhaps the biggest practical drawback: AC gives you a fixed plan. Once you start executing, you don't change the plan even if the market moves. In reality, if the price starts dropping fast, a trader might want to accelerate selling to avoid further losses; if the price is rising, maybe slow down to take advantage. AC's strategy can't do this because it's computed open-loop (no feedback from new price information). In modern electronic markets with fast-moving information, this inability to adapt is a severe limitation.

These limitations open the door for a more flexible, data-driven approach that can handle complex dynamics and adapt on the fly.

Motivation for RL: Given AC's shortcomings, we consider a **Reinforcement Learning (RL)** approach to optimal execution. RL is attractive here for several reasons: - It's **model-free**, meaning we don't need a perfect mathematical model of price dynamics or impact. The agent learns how the environment behaves by interacting with it (or via simulated experience). This is crucial because real market dynamics are complex and

partly unpredictable; a learning agent can adapt to whatever patterns exist in the data, rather than us hard-coding assumptions. - RL (especially with deep learning) can handle **high-dimensional data**. For example, we can feed the agent a rich state: recent price movements, order book signals, remaining inventory, time left, etc. AC, in contrast, only considered current time and shares left (and assumed a simple process for prices). - We can specify almost any **objective/reward** for the RL agent. We are not limited to mean-variance; we could directly penalize extreme losses, include transaction fees, or enforce various constraints by shaping the reward. This flexibility lets us target outcomes that matter most to us (like reducing tail risk). - Crucially, an RL agent produces a **policy** that is *feedback-based*: at each step it looks at the current state (including the latest price info) and chooses an action. This means the strategy is **adaptive**. If the price suddenly jumps or crashes, the agent can react in real-time by adjusting its trading rate. We're no longer stuck with a predetermined schedule — the policy is effectively doing what a human trader would do: observe and react, but in an optimized, learned manner. - Given these points, RL has the potential to outperform AC in scenarios where the market environment is complex or non-stationary, by learning and adapting to those complexities.

DDPG Overview: We chose an RL algorithm called **Deep Deterministic Policy Gradient (DDPG)** for our project. DDPG is well-suited here because our action (how much to trade) is continuous. Traditional RL algorithms like Q-learning struggle with continuous actions, but DDPG handles them elegantly by using a neural network policy.

Here's how DDPG works: - It's an **actor-critic** method. We have two function approximators (neural networks): the **Actor** and the **Critic**. The Actor takes the current state (market conditions, time, inventory, etc.) and outputs an action (in our case, a number between 0 and 1 representing how aggressively to trade). The Critic takes a state and an action and outputs a value (how good that state-action pair is in terms of expected cumulative reward). - The policy is **deterministic**, meaning for a given state s, the Actor always gives the same action $a = \pi(s)$ (no randomness inside the policy). This is different from some other algorithms that use stochastic policies. Because the policy is deterministic, DDPG needs a separate strategy for exploration during training; hence we add noise to the output actions when interacting with the environment to explore different possibilities. We use an Ornstein–Uhlenbeck (OU) noise process, which generates temporally correlated noise (this is often used in physical control problems to model smoother exploration). - Training is **off-policy** and leverages an experience replay buffer. Every time the agent interacts with the environ-

ment, the experience (s, a, r, s') is stored. The networks are then trained on random mini-batches sampled from this replay memory. This breaks the temporal correlations in training data and improves stability and sample efficiency. - We also use **target networks**: essentially we keep a slowly-updated copy of the Actor and Critic networks to compute target values in the Critic's update. This is a common trick to stabilize training (prevents oscillations that can happen if the target moves too fast). - The learning process: The Critic network learns by trying to satisfy the Bellman equation $Q(s, a) \approx r + \gamma Q(s', \pi(s'))$. We adjust the Critic's weights to minimize the mean squared error between its prediction and this target (computed using the target networks). The Actor learns using the Deterministic Policy Gradient: essentially, we adjust the Actor's weights to maximize the Critic's Q-value (intuitively, pushing the Actor to choose actions that the Critic thinks are high-value). This is done by backpropagating through the Critic with respect to the action. - Summing up: DDPG gives us a way to learn a **continuous trading policy**. It has been successfully applied in continuous control problems (like robotic control), and here we apply it to decide how to break up trades. However, as we'll see, DDPG is not perfect and can have stability issues, which will motivate us to explore its improved variants.

The agent's decision at any moment depends on the **state** we feed it. Designing this state representation is crucial, as it encodes what information about the market and the trade the agent can use.

Our chosen state s_k at time step k includes: - **Recent log-returns:** $r_{k-D+1}, \ldots, r_k$, which is basically a short window of the asset's price returns leading up to the current time. Here D is the window length (we used $D = 5$, meaning the state includes the last 5 log-returns). This sequence gives the agent a sense of recent **momentum** (is the price trending up or down?) and **volatility** (are returns choppy or stable?). For example, if all recent returns are negative, the price is consistently dropping — the agent might infer a downward trend and decide to sell faster. The reason we use log-returns (rather than raw prices) is to normalize and stationarize the input to some extent. - We did some experimentation with the window size D. A larger D (more historical returns) could in theory give the agent more information about patterns or mean-reversion, but it also increases the state dimension and the complexity the network has to learn. If D is too small, the agent might not detect momentum; if too large, the extra information might just be noise and make learning harder. Empirically, we found that a D in the range of about 3 to 8 gave similar results — not much gain beyond 5, so we stuck with 5 for our runs. - **Time remaining m_k:** We encode how much

time is left in the trading horizon as $m_k = (T-k)/T$ (assuming T is the total number of time steps in the episode). This is a normalized countdown from 1 to 0. Intuitively, this gives the agent a sense of urgency. At the beginning (m close to 1), there's plenty of time; near the end (m approaching 0), time is almost up, which should prompt the agent to finish selling whatever is left. Without a time signal, the agent might not differentiate early vs. late in the episode and could behave inconsistently with the deadline. - **Inventory remaining i_k:** This is the fraction of shares left to sell, Q_k/Q_0. It starts at 1 (100 of the order outstanding) and goes to 0 when we're done selling. This informs the agent of how far along it is in completing the task. If a lot of inventory remains (close to 1) and time is short (m small), that's a warning sign the agent is behind schedule and might need to sell more aggressively. Conversely, if only a little inventory remains early on, the agent might slow down to avoid unnecessary impact.

All these components together make the state. Notice they are all **normalized** (fractions or standardized returns), which helps training. The state is Markovian enough for our needs: given this information at time k, plus the agent's action, the next state captures everything relevant (we don't, for example, explicitly carry over the previous action, though the effect of previous actions is reflected in inventory left and price changes).

We define the action in a very straightforward way: it's a number between 0 and 1 representing how aggressively to sell at the current step relative to what's left. - If $a_k = 0.2$, that means "sell 20 of my remaining shares right now." - If $a_k = 1$, that's "sell everything that remains immediately." - If $a_k = 0$, that's "hold off, don't sell anything this period."

By defining the action as a fraction of remaining inventory, we ensure the action space is nicely bounded (the neural network outputs get clamped between 0 and 1) and we don't have to worry about the agent trying to sell more shares than it actually has. It's intuitive as well: the agent's output can be thought of as "what portion of what I have should I try to unload now?"

When the agent chooses a_k, we convert that into an actual share quantity: $Q_{\text{sell},k} = a_k \times Q_k$. So the volume sold in that step is directly proportional to how much is left at that moment. One nice property: as the inventory Q_k decreases over time, a given action a_k corresponds to a smaller absolute number of shares. This automatically makes the agent's behavior scale down as it nears completion (if it kept outputting say 0.5 each time, the absolute shares sold goes down as inventory shrinks).

Also, this formulation implicitly stops the agent from overshooting: - You can't sell more than 100 of what you have (a_k is at most 1). - If the

agent tried to be clever and output $a_k > 1$, we clamp it (and similarly we ensure no negative actions). So physically it won't go short or something bizarre.

We highlight that this is our primary action definition, but we also explored alternative ways to interpret the action output. For example, one alternative is to treat a_k as a **trading rate** rather than a fraction of remaining. In that scheme, a_k could correspond to, say, a fixed number of shares per unit time or a fraction of some benchmark volume. We even tried modifying actions based on market conditions (like widening spreads or volatility), which I'll touch on later. But the fundamental scheme remains: a continuous control between doing nothing and selling everything, which is a natural fit for an algorithm like DDPG.

Designing the Reward: In reinforcement learning, what you reward is what the agent will optimize. We intentionally crafted a reward function to encode our execution objectives: - The first term $-\mathrm{ES}_{0.95}(\mathrm{P\&L})$ is the negative expected shortfall at 95% confidence. In plainer terms, if $\mathrm{ES}_{0.95}$ is, say, \$1,000 (meaning in the worst 5 outcomes you lose on average \$1,000), then the reward contributes -\$1,000. The agent will try to make this term less negative, i.e. reduce that expected shortfall. This directly pushes the agent to avoid those nasty tail outcomes. It's like saying "I really care about how bad things can get; please make the worst-case losses as small as possible." We implement this CVaR term using an auxiliary variable technique (a common trick to make CVaR differentiable for RL: essentially, the agent outputs an additional parameter η to estimate the quantile, and we include a loss that approximates the CVaR via that η). - The second term $-\lambda_1 \sigma^2$ is a penalty on the variance of P&L (profit and loss). This adds general risk aversion, not just tail. By penalizing variance, we encourage consistency in outcomes — the agent will prefer strategies that yield more predictable costs. We chose λ_1 through some experimentation to balance this term relative to the ES term (so that both tail risk and overall variance matter). - The third term $-\lambda_2 \, \epsilon \, |Q_{\text{sell}}|$ is explicitly punishing the agent for trading volume, scaled by a fee ϵ. Here ϵ can be thought of as the per-share transaction cost (like commissions or bid-ask spread cost). If the agent trades a lot of shares, it accumulates fees (negative reward). So the agent is incentivized to avoid unnecessary trading — for example, it might not rapidly buy-sell-buy-sell, it will just execute the necessary sells. In our context, since we're only selling, this term is basically a constant times total shares sold (which is fixed Q_0 by the end), but it does matter when discounting is involved or if it had the option not to trade some shares (though here it must sell all by the end). The more immediate effect is it

provides a slight push to be efficient in trading. - We apply a very light time-discount ($\beta = 0.9999$). This means a reward (or cost) received one step later is worth 99.99 of what it would be if received now. It's almost undiscounted because our episodes are not very long (maybe hundreds of steps at most within a day). The reason we didn't use exactly 1.0 is that by using $\beta < 1$, we introduce a tiny preference for getting positive rewards (or incurring negative rewards) sooner rather than later. In execution terms, this encourages completing trades slightly earlier if all else is equal. It's a subtle bias to discourage procrastination in selling, thereby reducing the risk of being caught with inventory at the very end.

The combined reward is accumulated over an episode. A high total reward corresponds to a strategy that has low tail losses, low variance, and low fees. We flipped signs (using negatives) for ES and variance because we treat costs as positive "losses" and we want to minimize them (maximizing the negative of a cost is minimizing the cost).

In summary, this reward shaping is our way to encode the idea: "Complete the order with minimal cost, but more importantly avoid the worst-case scenarios." It's a more nuanced goal than AC's pure mean-variance; we've added a tail-risk focus and fees. Later, we'll compare how an agent trained with this reward stacks up against the AC strategy.

We also experimented with how the reward is given: **dense** vs. **sparse** reward structures. - In a **dense reward** scheme, we compute some reward at every step. For example, we might calculate the negative cost incurred by the trade in that step (or the improvement in some utility) and give that as immediate reward. This means the agent is constantly getting feedback: if it does something good in a particular time step (like sells at a favorable price), that step's reward will reflect it. - In a **sparse reward** scheme, we don't give the agent any reward until the very end of the episode. Only after it finishes the entire order (or at the end of the day) do we compute one cumulative reward (for instance, based on total implementation shortfall or our ES-based utility). Throughout the episode, the agent has to just figure things out without intermediate hints.

Trade-offs: - With dense rewards, learning is typically faster. The agent doesn't have to wait until the end to know if an action was good or bad; it gets a more immediate indication. In our case, when we tried giving a dense reward (like incremental negative cost per step), the training convergence was notably quicker. The agent can gradually learn, step by step, how its actions affect short-term outcomes. - However, dense rewards can sometimes make the agent chase short-term gains at the expense of the final objective. For example, an agent might learn to opportunistically grab

small immediate rewards (like selling a bit when price ticks up) even if that's not globally optimal for the end-of-day outcome. - Sparse rewards align perfectly with the final objective (since we only score the final outcome), so there's no ambiguity in what we want. But they make the credit assignment problem much harder: the agent only knows how it did after everything is done, so it's tricky to learn which specific actions were good or bad. This usually means you need a lot more training episodes for the agent to stumble upon good strategies and get meaningful feedback. - In our experiments, we indeed saw what theory suggests: with a dense reward, the agent's performance (learning curve) improved rapidly early on. With a sparse reward, initially the agent is pretty clueless (since it's basically trial and error for a long time) but eventually it can learn a strategy that sometimes is more nuanced or risk-aware. - Interestingly, we observed that agents trained with sparse final rewards sometimes exhibited more cautious behavior – likely because only the final outcome matters, they tend to focus on ensuring a good end result (even if it means small sacrifices along the way). The dense-reward agents, in contrast, sometimes found hacks to score points stepwise but those weren't strictly the best final outcome strategies (a kind of temporal overfitting to immediate reward). - In the end, we opted for a mostly dense reward approach (for faster training), but carefully shaped it (as described on the previous slide) to align with final objectives as much as possible. We also tried to validate that the policy indeed optimizes the final metrics we care about, not just the stepwise rewards.

We initially started with the same price process assumption as AC for our simulation environment: an **Arithmetic Brownian Motion (ABM)**, meaning each time step the price move is basically a random draw from a normal distribution (with perhaps some drift). However, we wanted to challenge our execution strategies with more realistic price dynamics: - First, we moved to a **Geometric Brownian Motion (GBM)** model. In GBM, the log-returns (as opposed to price changes) are normally distributed. This ensures that the price can't go negative (which is a flaw of an unrestricted ABM) and that if the price grows, the absolute volatility grows with it (percent volatility stays somewhat constant). We used the SDE $dS_t = \mu S_t dt + \sigma S_t dW_t$. If you discretize that, roughly $S_{t+1} = S_t \exp((\mu - \frac{1}{2}\sigma^2)\Delta t + \sigma\sqrt{\Delta t}\,\eta)$ for $\eta \sim N(0,1)$. Practically, this meant in our simulator we adjusted how we generate price moves: instead of adding a Normal(0, σ^2) to the price, we multiply price by exp(Normal($\mu\Delta t$, $\sigma^2\Delta t$)). We also calibrated σ so that one time-step volatility under GBM matched the ABM's, to make comparisons fair. GBM is a classic model for equity prices (the Black–Scholes model assumption), so it's a reasonable step up

in realism. - Next, we introduced **autocorrelation in returns** using an AR(L) process. Real market returns are not completely independent; for example, there can be short bursts of momentum or mean reversion. To model a simple form of this, we let the returns (or price changes) follow an autoregressive pattern. For instance, an AR(1) model: $r_t = \phi\, r_{t-1} + \epsilon_t$. If $\phi > 0$, you have momentum (a positive return tends to be followed by another positive return); if $\phi < 0$, you have mean reversion (a positive return tends to be followed by a negative return). We extended this to AR(L) to allow some longer memory (maybe $L = 2$ or 5, etc.). This means our price simulator might produce short trends or oscillations that an adaptive strategy could potentially exploit. - We also considered the fact that volatility itself in markets is not constant — periods of high volatility tend to cluster. While we didn't implement a full GARCH model in this project, we acknowledge that to truly mimic market behavior you'd include a volatility model. For now, the AR dynamics give the agent something beyond random noise to latch onto (some predictability), and we keep the volatility mostly fixed (except in GBM it scales with price level).

In summary, by moving from ABM to GBM to AR(L), we progressively relax the assumption of "i.i.d. returns". GBM introduces a realistic scaling of volatility, and AR(L) introduces dependency between returns. These more complex environments will test whether the AC strategy or our RL agent can cope with, or even exploit, these conditions.

The different price dynamics we introduced have notable effects on what the optimal execution strategy should do. Let's discuss how our learned RL strategy adapts versus what AC would prescribe: - **Volatility effect (GBM):** Under GBM, the volatility grows with the price level, which effectively means you can get bigger absolute moves than under ABM (especially if the price trends upwards, the dollar swings get larger over time). We found that the RL agent trained with a CVaR (tail-risk) objective became more cautious in a GBM setting compared to ABM. Specifically, it tends to **execute sooner (front-load)** more of the order if it perceives volatility is high. Intuitively, if prices are very volatile, there's a greater danger that holding inventory will result in a big adverse price move. The AC strategy, on the other hand, was derived assuming a fixed volatility; it doesn't adjust mid-course if volatility changes. In our simulations, if we suddenly had a volatility spike, AC's static plan would not react, whereas the RL agent could ramp up its selling rate to get out of the position quicker and avoid the heavy tail-risk. This behavior is exactly what we hoped the CVaR penalty would encourage – when the downside risk increases, the agent sacrifices a bit of expected cost to reduce the chance of catastrophe. - **Drift

(trend) effect – downtrend:** If the price has a negative drift (meaning it's expected to decline over the trading horizon), clearly the best action is to sell as early as possible. The AC model (if extended to include drift in the optimal control solution) indeed says "sell faster when $\mu < 0$." Our RL agent, not knowing μ explicitly but observing the price path, does pick up on a downward trend. For example, in an episode where the price starts ticking downward consistently, the agent's policy will increasingly favor large a_k values (close to 1) – essentially trying to unload quickly before things get worse. We saw this when comparing scenarios: in downtrending simulations, the RL agent's trajectory of selling was steeper (more front-loaded) than in flat or uptrending scenarios. AC's fixed plan for a given drift would front-load to some extent too, but if the drift changes mid-way or if there's a sudden drop, AC can't adjust whereas RL can. - **Uptrend or short-term momentum:** If the market is going up on average, there's an temptation to hold inventory longer to sell at higher prices. However, this comes with risk – the trend could reverse. Our RL agent struck a balance due to its risk-aware reward: it would not blindly hold everything to the end, but it would slightly **back-load** the execution when it sensed continued upward price movements. For instance, in an environment with positive drift or AR(1) momentum, the agent might initially use smaller a_k (sell slower) than it would in a flat market, thereby riding the rising price for a bit. But as time wears on (or if volatility is high alongside the drift), it starts selling to ensure it completes the order. Essentially, it "leaves some money on the table" by not holding to the very end, because the objective penalizes large downside risk too. AC's original solution (with zero drift) would have sold at a steady pace regardless of that early uptrend, thus missing out on some profit opportunity.

To sum up, our RL agent can dynamically tailor its execution based on observed market patterns: - In calm or no-trend conditions, it behaves similarly to AC (no obvious reason to deviate). - In adverse conditions (falling or very volatile market), it gets more aggressive than AC to reduce risk. - In favorable conditions (rising market, short-term predictability), it's willing to be more patient than AC, improving returns, while still minding the clock and risk.

These behaviors highlight the adaptability of the RL approach. In the next slides, we'll quantify how much improvement we get from RL under these various conditions.

Aside from the stochastic price dynamics, we also enriched our **cost model** in the simulator to be closer to reality: - We explicitly model **trading fees**. In many markets, there's an exchange fee or a bid-ask spread cost

for each trade. AC effectively had a term for a fixed cost per share (often denoted by ϵ in literature, representing half the bid-ask spread or some fixed slippage). We include that in our reward/cost calculations. Practically, it means if you sell x shares, you pay $\epsilon \cdot x$ in fees (so ϵ might be a few basis points of price, for example). This encourages the agent not to over-trade or churn. - More interestingly, we introduced a **non-linear impact cost** by adding a quadratic term in volume. Why? In real order books, if you execute a very large trade in one go, the price impact isn't linear. You start eating into worse prices as you consume the order book. A common approximation is a nonlinear impact function, often quadratic for simplicity (impact cost $\sim \gamma v + \kappa v^2$, etc.). In our case, we add $\lambda_2 v^2$ (as mentioned earlier in the reward). So if the agent doubles the trade size in a single step, the impact cost could quadruple (depending on calibration). This strongly punishes unreasonably large trades. - We observed that adding this quadratic term changes the behavior: The agent becomes much more averse to "dumping" a huge quantity in one shot. It tends to break trades into smaller chunks spread across consecutive time steps to avoid paying the heavy quadratic penalty. For example, without the v^2 term, the agent might occasionally try to sell a big chunk if it thinks price is about to plummet (taking the immediate impact hit but avoiding future loss). With the v^2 term, doing so is extra expensive, so the agent will still accelerate selling in a downtrend, but it will do so in a slightly more controlled manner (not literally everything at once, unless absolutely necessary). - In terms of overall performance: By penalizing jagged, spiky trading, we get a smoother execution trajectory, which is generally desirable (less market disturbance). If we compare utility or cost outcomes with vs. without the quadratic impact, strategies with the quadratic cost achieve a better cost-risk trade-off when you account for that realistic cost. It's basically teaching the agent: "if you try to eat too much liquidity too fast, the market punishes you severely, so it's better to slice the order more." - This adjustment also brings the simulation a step closer to reality, which means any strategy learned will likely behave in a liquidity-friendly way, an important consideration for deploying in actual markets.

Now let's talk about how the DDPG agent actually performed. We trained the agent in our simulation environment. Each training episode corresponds to executing one full order (say over a day broken into discrete time steps). Over many episodes, the agent gradually improved its strategy.

Training details: We used a replay memory of 10,000 past time-step transitions for learning, with mini-batches of 128 sampled for each update. The discount factor was $\gamma = 0.99$ (since our horizon is somewhat short, this

is fine, and we already bias towards early trading via reward shaping). We applied Ornstein–Uhlenbeck noise to the actions during training to encourage exploration (with parameters chosen such that the noise has a decent magnitude early on and decays over time). The actor and critic networks were relatively small (two hidden layers of 24 and 48 units respectively, using ReLU activations) – this size was sufficient given the state dimension is not huge, and we wanted to avoid overfitting to our simulator idiosyncrasies.

We monitored the training process: initially, the agent's performance was poor (worse than AC, basically flailing randomly). But after many training episodes, it started to learn a reasonable policy. The learning curve (if shown) would indicate the cumulative reward (which encapsulates our cost-risk objective) improving and eventually plateauing, meaning the agent converged to a stable policy.

Policy behavior: The learned policy qualitatively showed sensible behavior – for instance, in stable markets it would sell in a pattern not unlike AC's schedule; in volatile or trending scenarios, it adjusted as we described before. Importantly, it learned to manage the trade-off we imposed: it wasn't just minimizing cost, but actively avoiding scenarios where it would be caught with too many shares in bad conditions (due to that ES term).

Importantly, this tail-risk improvement did not come at the expense of a higher average cost; the RL's average cost was comparable to AC's (sometimes even slightly better, depending on environment). This means we essentially managed to *shift the distribution* of outcomes in a favorable way: fewer nasty outliers without paying more on average. This is a big win for practical trading, because it means a more reliable execution performance.

So, in summary, our DDPG agent outperformed the traditional AC strategy, especially on risk-sensitive metrics. This validates the approach of using RL with a custom objective – it found a strategy that AC (with its static, linear assumptions) couldn't match in our simulated market.

We tested the trained DDPG agent and the AC strategy under different simulated market scenarios to see how they compare: - **In a basic random walk scenario (ABM with no drift):** This is the scenario AC was built for. Not surprisingly, the AC strategy is nearly optimal here. Our RL agent, interestingly, ends up behaving quite similarly to AC in this case. Since there's no trend or pattern to exploit, the best you can do is something like "execute evenly across the horizon" (with slight front-loading if risk-averse). The RL agent basically re-discovered that solution because any deviation didn't improve its reward. Consequently, the performance metrics (average cost, risk measures) for RL and AC in a pure ABM environment were very close. This is good – it means at least the RL didn't do worse unnecessarily;

it learned that when the environment is exactly as AC expects, AC's solution is hard to beat. - **In a trending environment (GBM with drift):** Here we saw a divergence. We considered cases of both positive drift (bullish market) and negative drift (bearish market). AC's original formulation doesn't explicitly include drift (unless we modify it), so if one naively used the AC schedule tuned for zero drift, it's not ideal under drift. Even if we gave AC knowledge of drift, it would commit to a single plan at the start. The RL agent, on the other hand, adjusts as it goes. In simulations with a downward drift, AC (with no drift adjustment) might spread out selling and end up holding too long as price falls — incurring a large cost by the end. Our RL agent recognized the downward trend and sold much earlier, resulting in a lower implementation shortfall. In upward drift cases, AC might sell too quickly (missing out on the rising prices), whereas RL learned to pace itself more and capture better prices on average. The result: RL had better performance (lower cost) in both uptrend and downtrend scenarios compared to a non-adaptive AC. If we gave AC the drift in advance and recalculated an optimal strategy, AC would do better than the naive case, but still, any unexpected acceleration or deceleration in the trend mid-way can't be handled by AC's fixed plan, whereas RL can adjust on the fly. - **In environments with autocorrelation (AR(L) returns):** These are scenarios with short-term predictable patterns – for example, maybe the price often alternates up and down (mean-reversion), or it tends to continue in the same direction for a few steps (momentum). AC's strategy does not utilize any such information; it's oblivious to serial correlation. The RL agent, however, has those recent returns in its state and can learn to respond to them. Indeed, we observed that the RL agent could exploit momentum: if it "senses" an upward price momentum, it might hold back a bit and wait for a slightly higher price to sell. Or in a mean-reverting scenario, if the price just dropped, RL might anticipate a bounce-back and avoid panic-selling at the bottom, whereas AC would have sold a fixed amount regardless. These nuanced tactics let RL reduce cost – for example, selling after a small rebound instead of during a dip means a better execution price on those shares. Over many simulations, this translated to a noticeable cost saving vs. AC, which would have sold into the dip without second thought. - The takeaway is, **whenever the market has structure or patterns that AC doesn't account for, the adaptive RL agent can pick up on those and improve performance**. In our tests, the more structured the environment (like strong drift or autocorrelation), the larger the improvement by RL. In a completely structureless environment (pure noise), RL and AC were on par. This aligns with intuition: an adaptive strategy can't beat a static one

if there's nothing to adapt to (only random noise), but it shines when there is something exploitable in the price path.

We performed a range of ablation experiments to understand how different design choices affected the agent's performance. This includes how we map the action to actual trading and how we shape the reward.

Alternate Action Mappings: Our base action definition was straightforward (fraction of inventory). But we thought: can we inject some domain knowledge by transforming the agent's action in certain ways? - In a "spread-adjusted" approach, we tried to account for the bid-ask spread and recent price momentum. If the spread is wide (market is illiquid or expensive to trade) and price momentum is negative, maybe reduce trading, for instance. The specific formula we used was something like $a_k^{\text{effective}} = a_k \times [1 + (\text{rel. spread}) \times (\text{last return})]$. The idea was to encourage the agent to trade more when momentum is positive (price going up) and less when momentum is negative, modulated by spread (which indicates cost of trading). The rationale is if the price is moving in your favor (up) and spread cost is not too bad, you can sell a bit more now because you're getting a good price; if price is dropping (momentum negative), selling into that might lock in a worse price, especially if spread is high, so perhaps hold off a bit. - The "volatility-adjusted" action scaling was aimed at risk management: when recent volatility is high, we scaled down the agent's action. For example, if the last few returns had a standard deviation double the normal level, we might cut the effective action in half. This means in turbulent times, the agent automatically trades less per step (to avoid transacting in the middle of wild swings, possibly waiting for calmer periods or at least not piling on impact in an already volatile moment). - The "volume-constrained" mapping enforced a rule that we never trade more than a certain percentage of the overall market's daily volume (we set something like 1 of daily volume per time step, which is a common heuristic in industry to minimize market impact). So even if the agent wanted to sell a lot, we cap it. If the agent output a_k times inventory would exceed that cap, we limit it. This ensures the strategy is realistic — no 50 of daily volume in one minute kind of moves. - Other variations included defining a_k as a fraction of initial total shares (which effectively is like a TWAP if constant) or as a continuous trading rate (like $Q_{\text{sell}} = a_k \times (Q_{\text{remaining}}/\text{time\_remaining})$). These were more experimental and in practice similar outcomes to the fraction-of-remaining baseline or the time-based scaling.

Reward Tuning: We also toggled aspects of the reward: - We tested a more traditional mean-variance reward (like $-\text{Cost} - \lambda\text{Var}$ without the CVaR term) to see how the agent behaves. As expected, without the CVaR

term, the agent focuses on reducing variance but might occasionally incur a big tail loss because it's not explicitly optimizing the tail. The strategies were a bit less conservative – slightly lower average cost but higher tail risk than the CVaR-optimized version. - We already discussed dense vs sparse rewards. We confirm that the dense reward agents learn faster but sometimes at the risk of focusing on short-term rewards. The sparse reward agents learned more slowly but their eventual policy was slightly more aligned with final outcome optimization. In practice, we stuck with mostly dense reward training (for efficiency) but tried to design the dense reward to approximate the final objective well (so the agent doesn't go off-track).

Impact on behavior: - The presence of the CVaR term in the reward definitely made the agent avoid risky end-of-horizon situations. For example, a CVaR-trained agent would rarely end an episode with a large fraction of inventory unsold, because that scenario can lead to really bad outcomes in some trials (if the price suddenly tanked at the end). An agent without CVaR might occasionally cut it closer to the deadline with inventory left if on average it seemed beneficial. - Agents with dense rewards tended to make more aggressive early moves. They got rewarded immediately for early good trades, so they sometimes traded a lot early if it meant immediate profit, maybe not fully accounting for later risk. In contrast, the sparse agents often waited a bit (since there's no immediate penalty for waiting except at the very end). The sparse ones acted more like "nothing matters until the finish line," which sometimes led to sudden large trades near the end. They were very cautious in the beginning and then had to rush later to complete — which can be risky too. So there's a balance. - The volatility-scaled action mapping turned out to be quite helpful. It injected a kind of sensible risk management behavior: during high volatility periods, even if the agent's raw policy said e.g. 0.5 (sell half), the scaling might reduce that to 0.3, meaning it trades a bit less when things are crazy. This tended to reduce variance in outcomes — basically, not dumping a huge amount in the middle of a volatile swing often avoided selling at a locally bad price. The agent then might catch a slightly better price a few steps later when volatility subsides or price mean reverts. - The volume cap obviously prevented extreme trades. Interestingly, with that in place, the agent learned to operate within those bounds pretty naturally. It basically never attempted a trade that would violate the cap once it learned, because presumably it figured out that trying to do so just saturates the cap and yields no extra benefit. This made our results more realistic and stable (no outlier episodes where the agent does something that would be impossible in real markets).

Best Configuration: After all these tests, we identified a combina-

tion that worked best for our objectives. The top performer was the agent trained with the tail-risk (CVaR) reward on a dense schedule, using the volatility-adjusted action mapping (plus the volume cap for realism). This configuration achieved the lowest 95 ES and a very good average cost. Essentially it maintained the low tail risk we wanted, and still kept the average cost close to minimal. We'll fill in the exact numbers in the report, but qualitatively, it might have improved ES by, say, 20-30

In short, these experiments showed us that both the reward design and action parameterization have a significant effect. By choosing them wisely, we got a noticeably better outcome. It also demonstrated the flexibility of the RL approach: we can tweak what we ask the agent to do (via rewards) and how it can act (via action mappings) to guide it toward strategies that align with trading intuition and practical constraints.

While DDPG worked for us, it's important to acknowledge its shortcomings, especially as found in literature and some signs we saw in our project: - **Overestimation Bias:** This is a known issue where the critic network in DDPG (and similar algorithms) can overly estimate Q-values. Because the actor is trying to maximize the critic's Q, if the critic has any consistent overestimation, the actor may exploit that, focusing on state-action regions the critic (wrongly) thinks are great. In execution terms, this could manifest as the agent being overconfident about a strategy — for example, the critic might overestimate the reward of waiting longer to sell (because maybe in the training experiences, it saw a few lucky outcomes), so the actor might start waiting too long, which is not actually optimal. We did notice occasional oscillations in the critic's estimates and had to ensure good training practices to mitigate this (like plenty of training, even tried increasing critic updates). - **Training Instability:** DDPG is somewhat finicky. We had to tune learning rates (we used 10^{-4} for actor, 10^{-3} for critic) and the tau for soft updates, etc. If these were off, the training could blow up (e.g., Q-values diverging to huge values, or the policy collapsing to always 0 or always 1 actions). For instance, early on we had a scenario where the agent would suddenly start outputting $a = 1$ all the time — it turned out the critic had an issue and the actor exploited it. Solving this required adjusting parameters and occasionally resetting some components. This sensitivity means DDPG might not be the most robust if not carefully monitored. - **Exploration Difficulty:** Because the policy is deterministic, the only exploration comes from adding noise. We used OU noise with certain parameters. If the noise is too low, the agent might not try different strategies enough and gets stuck doing something suboptimal (like maybe it discovers a decent solution and just sticks around it without seeing if a drastically different approach

could be better). If the noise is too high for too long, the learning gets very noisy and could destabilize (the agent keeps thrashing, making it hard to learn Q accurately). We had to schedule the noise decay appropriately. In an execution context, exploration is tricky because a truly "wild" strategy might incur huge costs; thankfully in simulation that's okay, but if this were live or on real data, you'd want to constrain exploration. - **Sample Inefficiency:** We are training in a simulated environment, so we could generate many episodes. But if one were to use limited historical data to train, DDPG might need a lot of it. It doesn't learn super fast from small samples, partially due to the above issues. We had to run quite a lot of simulated episodes for convergence. This raises a question: if the environment were more complex or if data was scarcer, DDPG might struggle.

These limitations motivated us to try more advanced RL algorithms that address some of these problems (like overestimation and stability). In particular, algorithms like TD3 and SAC were designed to fix exactly these issues in continuous control tasks.

Introduction to TD3: **Twin Delayed DDPG (TD3)** is an enhanced version of DDPG proposed by Fujimoto et al. (2018) to address exactly the issues we just discussed with DDPG. It adds three key improvements: - First, TD3 employs **two critic networks**. In training, each time we want to compute the target Q-value or update the actor, we use the smaller of the two critics' estimates. This is like having a second opinion that keeps the first in check — by taking the minimum, we avoid the case where one critic wildly overestimates a value. This idea is analogous to Double Q-Learning used in the DQN world to curb overestimation. The result is more accurate (or at least not overly optimistic) value estimates, which in turn means the actor doesn't get misleading cues to exploit. In our project, implementing twin critics was straightforward, and it immediately helped in testing – we saw less bias in Q estimates. - Second, TD3 uses **delayed policy updates**. In DDPG, we update the actor and critic every time step (or every batch). In TD3, we might update the critic networks at every step, but only update the actor (and target networks) once every n steps (typically $n = 2$). This means the actor spends more time catching up to a stable evaluation from the critics rather than constantly chasing a moving target. It prevents the ping-pong effect where actor and critic keep overshooting each other. For us, this meant the policy changed more gradually, which we found gave a smoother training dynamic. - Third, **target policy smoothing**: When calculating the target Q-value in the Bellman update (i.e., $r + \gamma Q_{\text{target}}(s', \pi_{\text{target}}(s'))$), TD3 adds a small random noise to the action $\pi_{\text{target}}(s')$ and clamps it within the valid range. Essentially,

instead of using the target policy's exact action, we use a slightly "noisy" action. This trick prevents the critic from learning to exploit fine-grained unrealistic precision in the actor's outputs. In DDPG, if the actor finds a particular action that the critic thinks is super-high-value, it will zero in on it. But that could be a fluke or a narrow peak (maybe due to function approximation quirks). With smoothing, we're telling the critic: assume the actor's chosen action might vary by a small amount; the value should be somewhat robust in that neighborhood. This results in a critic that doesn't have extremely sharp peaks in the Q-function, making it harder for the actor to accidentally drive itself to a fringe of the action space based on a spurious Q spike. - Combined, these changes made a big difference. In our experiments, when we switched to TD3, the training became much more stable. The learning curves had fewer spikes and drops. We did not witness the same kind of sudden divergence that we occasionally saw with DDPG. The final policy performance of TD3 was on par with or slightly better than DDPG's best (depending on the scenario), but crucially it was reproducible and reliable. For example, if we train multiple agents with different random seeds, TD3 results tend to cluster closer together, whereas DDPG runs might have one great run and one that got stuck suboptimally. - In summary, TD3 inherits the advantages of DDPG (continuous actions, actor-critic structure) but mitigates its pitfalls. For an optimal execution task, this means we can trust the learned strategy more and not worry that a slight hyperparameter change will make it fail. It gave us confidence that the improvements we saw from RL are not just luck from one training run but can be consistently achieved.

The **Soft Actor-Critic (SAC)** algorithm is another state-of-the-art RL method we applied. It differs from DDPG/TD3 in a fundamental way: it uses a **stochastic policy** and optimizes for a mix of reward and entropy. - In SAC, the policy (actor) doesn't output a single action, but rather a probability distribution over actions. Typically, we model this as a Gaussian distribution for continuous actions. So given a state, the actor might output a mean and standard deviation, and then we sample an action from that Gaussian. This means that even if you present the same state multiple times, the policy could choose slightly different actions each time. Initially, these distributions are broad (lots of randomness), and as training goes on, they might tighten up if certain actions are clearly better. - The core idea of SAC is the **maximum entropy principle**. Instead of just maximizing expected return, SAC maximizes expected return *plus* a term for entropy (with some weighting). Intuitively, it tries to find the best reward it can get while still "being as random as possible." Why would we want that?

Because a high-entropy policy is more exploratory and doesn't prematurely converge to a possibly local optimum. It also tends to be more robust — if there are multiple good actions in a state, a deterministic policy might arbitrarily pick one, whereas SAC will essentially keep a repertoire (a distribution) of actions, which is helpful if conditions change. - SAC uses **twin critics** as well (so it also benefits from the overestimation bias reduction). It has an additional parameter α (the temperature) which controls how much randomness (entropy) is valued relative to reward. Too high α means it'll act almost randomly just to be "creative," too low and it behaves more greedily like a regular actor. One nice feature is SAC often implements an **automatic tuning** of α: it adjusts α during training to hit a target entropy (basically letting the algorithm decide how stochastic it needs to be). - In practice, we found SAC's training to be very stable. The inclusion of the entropy term seems to act as a regularizer — the policy doesn't move in extreme ways because it's always considering "I shouldn't reduce my entropy too much just to get a slightly higher reward." So you don't see oscillations; things converge smoothly. - For the execution problem, one might ask: do we want a stochastic policy? After all, for execution, a trader usually wants a clear plan, not a random one. However, there are a couple of perspectives: - During training, the stochastic policy is a huge benefit for exploration. The agent naturally tries a variety of actions, which helps it discover better strategies. - The final learned SAC policy can still be used in a deterministic way by taking the mean of the distribution as the action (or sampling if you prefer some randomness in execution to avoid being too predictable in the market). So it's flexible. - Also, from a game-theoretic perspective, a stochastic execution strategy can be harder for others to exploit or anticipate — but that's beyond our scope; in our case, we don't model other strategic traders, but it's an interesting side note that SAC would inherently give a randomization to your execution pattern. - Comparing performance, SAC achieved at least as good results as TD3 in our tests. It sometimes slightly edged out TD3 in terms of final reward, especially in scenarios with a lot of stochasticity, because its exploration is better. The strategies it learned were effectively similar to TD3's in expectation, but with some randomization. - One thing we carefully monitored was the entropy term: we didn't want it to keep the policy too random by the end (since unnecessarily random execution could add cost). With automatic tuning, we set a target entropy (often a small negative value per action dimension) so that the policy retains some randomness but not too much. The result is a policy that, for instance, might usually sell, say, 30 of inventory at a certain point but sometimes 25 or 35 — that kind of slight variability. - Ultimately,

SAC provided another level of robustness. It's considered one of the most reliable algorithms in continuous control, and our experience matched that: we could run SAC training multiple times and get very consistent outcomes with minimal hyperparameter fuss (beyond the α tuning which it handles itself).

A bit about how we actually implemented and tested TD3 and SAC in our project: - We didn't have to start from scratch; we extended our DDPG code. We kept the same neural network architecture (actor and critic hidden layers, etc.) for both TD3 and SAC to ensure a fair comparison. This means any differences in performance come from the algorithmic improvements, not from giving one algorithm a larger network or more capacity. - For **TD3**, the main changes in code were: - Creating two critic networks instead of one. During training, we compute losses for both critics (against the same target value) and update both. - We changed the actor update schedule: the actor (and target networks) update only once every N steps (we used $N = 2$). This was easy to implement with a simple counter. - When computing the target Q for the Bellman update: we take the next state, get the target actor's action, add a small random noise (clamped to, say, 0.2 range), and then feed that into both target critics, and take the minimum of the two Q values. That's our target. This was a key part to implement correctly to get TD3's benefits. - For **SAC**, the changes were a bit more involved: - The actor now outputs two values (mean and log-std of a Gaussian) for each action dimension instead of a single action. We sample an action using the reparameterization trick (sample a standard normal, multiply by std, add mean, then push it through a tanh function to squash it into [-1,1] and scale to [0,1] since our action is bounded). We had to ensure numerical stability when computing log-probabilities of actions (for the entropy term). - We still use two critics in SAC. The loss for critics is computed differently: it uses not just reward + $\gamma \min(Q_1, Q_2)$ as in TD3, but also subtracts the entropy term (so target $= r + \gamma[\min(Q_1, Q_2)(s', a') - \alpha \log \pi(a'|s')]$). We implemented that carefully. - We implemented the automatic temperature tuning: basically have a loss that tries to drive the mean entropy towards a target value. If the policy is too deterministic (entropy low), this loss will increase α to encourage more randomness; if it's too random, it will decrease α. Over time it finds a balance. - We chose a target entropy roughly like $-0.2\times$ (action dimension) (just an example, meaning we want a moderate amount of randomness). - We trained TD3 and SAC agents on the same types of episodes as the DDPG agent for a fair comparison. They saw the same sequence of environment scenarios in training (we even used the same random seeds for environment generation where possible to

ensure comparability of learning experiences). - Observations: - The TD3 agent's training curve was very smooth. It improved steadily and plateaued nicely. We didn't see the spikes in Q-values or the actor output saturating as sometimes happened in early DDPG trials. - The SAC agent's training was also smooth, albeit computationally a bit heavier per step (since it samples actions and computes entropy). It may have taken slightly more episodes to converge to the optimum because it's balancing entropy, but once it did, it was rock-solid – no sign of collapse or anything. - In evaluation, both TD3 and SAC agents performed on par or better than the DDPG agent. For instance, in one test scenario, DDPG achieved, say, an $ES_{0.95}$ of some value, TD3 might achieve a slightly lower (better) value, and SAC might achieve the best. The differences weren't huge (DDPG was already doing well), but the advanced algorithms gave us that extra edge and peace of mind about stability. - Especially noteworthy: the SAC agent had the consistently lowest tail losses in out-of-sample stress scenarios (like a very volatile day). This is likely because its training explicitly sought a high-entropy robust policy, which might generalize better to rare scenarios. TD3 was a close second, and both outperformed DDPG in worst-case metrics, though all three improved substantially over AC. - By implementing these, we validated that our approach isn't tied to a particular algorithm's quirks. The idea of using deep RL for execution holds up across DDPG, TD3, and SAC, with SAC/TD3 giving us more confidence in the results due to their improvements in training methodology.

Let's wrap up by comparing all the approaches and giving a recommendation: - **AC vs RL (general):** The Almgren–Chriss model is elegant and gives you a closed-form solution – which is great for intuition and quick computations. If the world behaved exactly as AC assumes (random walk prices, linear impact, etc.), you wouldn't need anything else. But in reality, we know the environment is more complex. AC's rigidity (one pre-planned trajectory) can lead to missed opportunities or risk mismanagement if something unexpected happens during execution. Our experiments confirmed that in more realistic scenarios (with drift, momentum, higher volatility), the AC strategy was suboptimal compared to what an adaptive strategy could do. So, while AC is a good *benchmark*, it's not the endpoint for an actual trading system dealing with real markets. It's akin to an idealized solution. - **DDPG:** introduced the ability to adapt and optimize for a complex goal (like tail-risk minimization). It demonstrated clear improvements over AC in our tests – especially in tail-risk reduction. DDPG essentially "learned" to do what AC cannot: react to the market on the fly. The drawback is that DDPG itself is a bit of a black box and not al-

ways stable to train. So, if one were implementing this in practice, you'd have to be careful to validate and stress-test the learned policy, and maybe retrain periodically. - **TD3:** we can think of as "DDPG 2.0." It gave us the benefits of DDPG (adaptivity, good performance) without some of the headaches (training issues). It's deterministic like DDPG, which means the strategy it produces is a fixed mapping from state to action – that's often desirable because it's straightforward to execute and analyze. In our results, TD3 was very reliable; if someone were to reproduce our project or take it further, TD3 would likely be a default choice because of its stability. - **SAC:** provided another perspective with its stochastic policy. Among the methods, SAC had the strongest learning stability and was least likely to overfit weird quirks. Its policies also inherently provide some randomness which, as I mentioned, could be a good thing (it's harder for others to anticipate your exact trades if you're not 100 deterministic). In terms of performance, it either matched or slightly exceeded TD3 and DDPG in our cost and risk metrics. The difference wasn't huge, but SAC often came out on top, especially under tricky conditions. If one doesn't mind a stochastic strategy or is okay with using the mean of SAC's policy for actual execution, SAC is an excellent choice. - **Recommendation:** In light of all this, our recommendation for a real-world optimal execution system would be to leverage these deep RL methods. One could start by calibrating an AC model for a rough plan (because it's quick and gives you a baseline schedule), but then deploy a policy refined via RL (like SAC or TD3) that can adjust around that baseline as market conditions dictate. We'd specifically lean towards SAC for its robustness; it gives confidence that the agent won't do something crazy under novel conditions (because of the entropy regularization – it tends to continue trying reasonable exploratory moves rather than deterministic extremes). - AC isn't thrown away – you'd still use it to sanity-check ("is the RL suggesting something drastically different from AC? If so, is there a good reason?") and possibly as a fallback if the RL model is uncertain. - In scenarios that exactly match AC's assumptions (which might occasionally happen in very liquid markets with no trends), the RL agent effectively behaves similarly to AC, so you're not losing anything by using RL. But in scenarios where AC's assumptions break, the RL agent gains a clear edge, reducing costs and risks. - Additionally, our exploration of multiple algorithms showed that the superiority of RL is algorithm-agnostic to some extent – both value-based (TD3/DDPG) and policy-based (SAC) methods improved upon AC. This gives confidence that the improvement is due to the approach (learning from data and being adaptive) rather than any one algorithm's magic. - In closing, the project demonstrates a progres-

sion: we started from a classic model (AC), saw its limitations, introduced a learning-based solution (DDPG) to overcome those, and then refined it with modern RL techniques (TD3, SAC). The end result is a robust execution strategy that adapts to market conditions and outperforms the static AC benchmark, particularly in volatile or trending markets, all while managing risk (tail-risk specifically) more effectively.