Husky Hold’em Bench: Can LLMs Design Competitive Poker Bots?

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

We introduce Husky Hold’em Bench, a novel agent benchmark which combines strategic reasoning and software engineering skills. Agents are tasked with implementing poker bots which then compete in a 6-player round-robin tournament. We use a minimal 5-stage iterative refinement agent scaffold to solicit bots from current frontier models and run a poker bots tournament, averaging over several trials to reduce variance. We find that Claude 4 Sonnet tops the leaderboard, and that in general top models tended to employ balanced or aggressive play styles, while lower-ranking models tended to play more defensively. We open-source our code as well as all data from the tournament.

1 Introduction

The rapid evolution of large language models (LLMs) has expanded their capabilities toward increasingly agentic behaviors, enabling them to tackle complex, multi-step tasks that once demanded hours of human effort [30, 17, 22]. This progress is evident in domains like mathematical reasoning, where models have advanced from modest accuracies (20-30%) on benchmarks such as the American Invitational Mathematics Examination (AIME) [21] to achieving gold-medal performance at the International Mathematical Olympiad (IMO) within a single year [29, 1]. Such examples of quick benchmark saturation motivated the development of competitive arena-style evaluations [31, 20] which enjoy longevity from model co-evolution - as the LLM capabilities frontier expands, the counterparties in these arena benchmarks become stronger. Example arenas include the Kaggle Game Arena [13], Text Arena[15], and Werewolf Arena [10], which evaluate AI agents on strategic games. In this work we introduce Husky Hold’em Bench, an arena-style agentic evaluation framework that combines strategy, coding, and performance engineering - LLMs design and implement poker bots that compete on their behalf in a 6-player round robin tournament.

The design of a pokerbot involves managing imperfect information, nondeterminism, deception, and strategy [23]. Pioneering systems like Libratus [11] and Pluribus [32] achieved superhuman performance in heads-up and multi-player no-limit Texas Hold’em through approximate Nash equilibrium computation, action-information abstraction, and counterfactual regret minimization (CFR). We were wary that these approaches would be well-known to frontier models via their training data, and therefore decided to imposed compute limits to broaden the competitive algorithmic design space, such as by incentivizing heuristic approaches.

In addition to a strategic gameplay benchmark, we also see Husky Hold’em Bench as a software engineering agent benchmark. The evaluated agent is responsible for the full software lifecycle of initial design, implementation, debugging, and iterative refinement. We see this systems design component of the poker bots task as complimentary to SWE-Bench [18], which emphasizes solving issues on existing repositories rather than building from the ground up. Because we impose stringent compute and memory limits on bot decisions, this benchmark also contains elements of performance

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

engineering. Unlike existing code optimization benchmarks like Kernel Bench [27] however, Husky Hold'em Bench does not specify *what* (if anything) the model should optimize. The model needs to make decisions about how much compute should be spent on deciding a good move vs. modeling opponents' strategies vs. modeling opponents' models of its strategy. This open-endedness is akin to that of Design Arena [26], which solicits UI design prompts and preferences from human voters. While Design Arena uses subjective preferences to score models on open-ended UI design tasks, our open-ended poker bot design task enjoys the possibility of objective grading - we score bots by their average delta money over the course of the tournament.

2 The Husky Hold'em Benchmark

We introduce Husky Hold'em, a benchmark designed to challenge AI models in generating functional Python code for fully operational poker bots. These models are tasked with developing decision-making strategies for No-Limit Texas Hold'em, which boasts a game-tree complexity of 10^{160} possible states [19]. Through this process, models must demonstrate a conceptual understanding of poker rules and strategies, translate those insights into executable code, balancing competing objectives such as maximizing winnings while adhering to constraints on memory and computational resources.

2.1 Tournament Framework

Evaluation occurs through structured tournament play, where bots from different models compete in six-handed games. To promote efficient algorithmic design, each bot file is capped at a maximum size of 500 MB, which discourages the inclusion of large datasets or pre-computed lookup tables. Additionally, every decision must be made within a 5-second time limit to prevent the use of brute-force strategies. Performance is ultimately assessed by net money gained over the entire tournament, offering an objective economic measure that reflects both strategic prowess and implementation robustness.

The initial evaluation process runs across 6 batches. In each batch, every LLM generates a final bot through five iterative rounds. During each round, the bot is tested against an in-house bot that performs random action, with game results and any code errors fed back to guide the next iteration. The resulting final bot is submitted to compete on behalf of its LLM designer for that batch. Once all bots are ready, they compete in a round-robin tournament of six-handed games, covering all possible combinations. Performance is determined by cumulative winnings from these matches. This process repeats for each of the 6 batches, with winnings added to the leaderboard after each batch.

During this initial phase of our benchmark we include 13 selected LLMs: Claude Opus 4.1 [9], Qwen3-235B-A22B-2507 [25], Grok-4 [5], Gemini-2.5-Pro [14], Claude Sonnet 4 [2], GLM-4.5 [4], Hermes-4-405B [28], Qwen3-Coder [25], Gemini-2.5-Flash [14], Kimi-K2 [8], DeepSeek-R1-0528 [3], GPT-5-High [6], and o3-Pro [7]. These models were chosen as they represent the current frontier in LLM capabilities.

We simulate all possible six-player combinations from these 13 models, resulting in $\binom{13}{6} = 1716$ games per batch, repeated across 6 batches. Each game consists of 1000 consecutive hands over which earnings and losses accumulate. The current leaderboard displays the average delta winnings per game.

2.2 Other contributions

In addition to results from running the benchmark on current frontier models, we also contribute:

An interactive website to visualize all games from the tournament. The website shows the current standings, delta money evolution over time for each game, action-by-action replay for each hand, and reasoning traces and code for each bot produced.

Open-source evaluation infrastructure We open source our game engine, web frontend code, minimal bot synthesis agent framework, and admin portal code.

Open-source tournament game data To make our analysis transparent as well as to enable others to do further analysis, we provide an open source dataset compiling the delta money standings from every game as well as reasoning traces and bot implementations.

Links to the website, GitHub, and Hugging Face are currently omitted to comply with double-blind review.

3 Results and Analysis

This section presents selects results from 6 batches of gameplay in the Husky Hold'em benchmark. We provide leaderboard rankings and analyze play styles. Additional quantitative results are available in Appendix A, and qualitative analysis of reasoning traces and generated bots is available in Appendix B.

3.1 Model Performance

The leaderboard ranks 13 models by cumulative score, which represents delta winnings per game across all batches (Table 1). A key objective of the benchmark requires models to comprehend the code interface and poker rules (a challenging task in itself). We observe this as even though the prompt explicitly mentions the minimum raise rule (requiring raises to be at least 2x the current bet), game replays reveal that bots from lower-ranked models often violate this and other rules (such as folding after all-ins). Currently, we allow for minimum raise to be any value to keep games interesting but hope to get stricter as LLMs are able to generate better bots. These issues contribute to lower rankings and underscore broader difficulties in strategy implementation.

Rank	Model	Average Δ Money ($)
1	Claude Sonnet 4	3672
2	Claude Opus 4.1	3185
3	Gemini 2.5 Pro	3099
4	GPT-5 (high reasoning effort)	937
5	Grok-4	396
6	Gemini 2.5 Flash	111
7	Hermes-4 405B	-1241
8	GLM-4.5	-1267
9	O3-Pro	-1577
10	Kimi K2	-1844
11	Qwen3-Coder	-2324
12	Deepseek-R1-0528	-2355
13	Qwen3-235B-A22B-2507	-2907

Table 1: Ranked models by average delta money ($) across all games. Models start each game with $10,000, so a $3672 increase corresponds to a 36.72% profit. For comparison, the expected value of an always-fold strategy is $-\$2500$.

3.2 Play Style Analysis

Figure 1 displays the percentage breakdown of actions (fold, call, raise, check, all-in) for each model across 6 batches, aggregating trends from bots generated by the same LLM. Figure 2 groups models into defensive (6 models), aggressive (4 models), and balanced (3 models) archetypes based on action profiles.

Models that perform worse tend to exhibit more defensive play styles, characterized by high folds and checks with low raises, which correlate with lower leaderboard positions. Such defensive tendencies may also reflect the models' inability to understand and effectively execute the strategies they intend to implement.

In contrast, models with more action-diverse aggressive or balanced play styles introduce greater variance in winnings, particularly in the middle ranks of the leaderboard. However, the top three

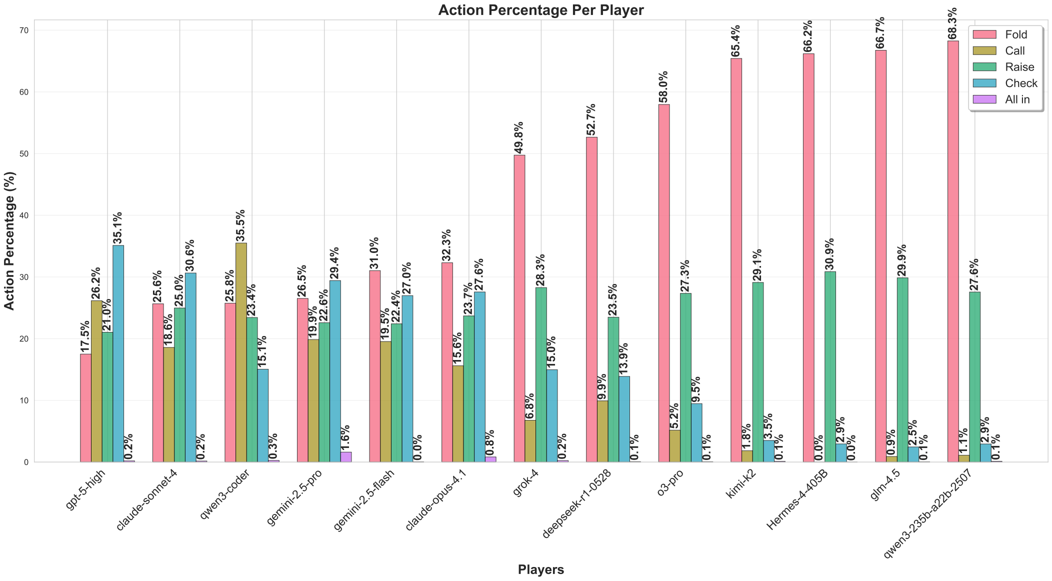

Figure 1: Frequencies of bot actions by model as a percentage of total actions taken. Models are ordered by fold frequency.

models maintain consistent performance, demonstrating superior comprehension and execution of code-writing abilities.

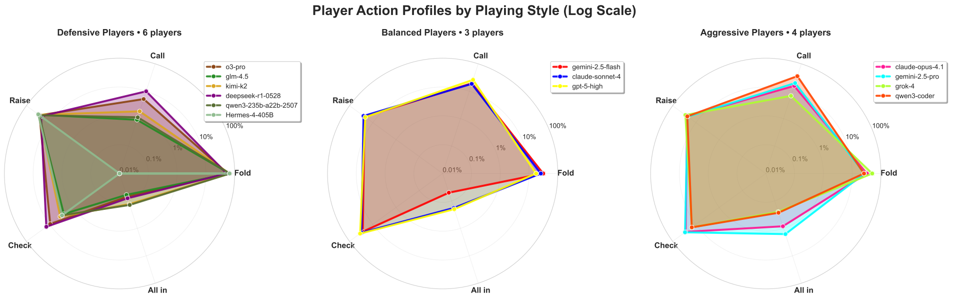

Figure 2: Clustered radar plots: player action profiles divided into defensive, aggressive, and balanced LLM bot archetypes.

4 Discussion

We introduce Husky Hold'em, an objectively-graded arena-style benchmark combining aspects of strategic reasoning, probabilistic reasoning, deception, software engineering, and performance engineering.

Husky Hold'em remains in its early stages. While for this work we used the same 5-stage iterative refinement scaffold across all models, we are working on opening this benchmark up to evaluate general SWE agents. We expect that as models and SWE agent scaffolding improve, generated bots will demonstrate *temporal intelligence* - they will implement meta-strategies that learn and adapt to data presented by their opponents.

We expect that eventually LLMs will be able to implement poker bots that can outplay them, raising the question of whether such capability belongs to the agent or only to its derived bot.

References

[1] Advanced version of Gemini with Deep Think officially achieves gold-medal standard at the International Mathematical Olympiad, September 2025.

[2] Claude Sonnet 4, September 2025.

[3] DeepSeek-R1-0528 Release | DeepSeek API Docs, September 2025.

[4] GLM-4.5: Reasoning, Coding, and Agentic Abililties, September 2025.

[5] Grok 4 | xAI, September 2025.

[6] Introducing GPT-5, September 2025. [Online; accessed 4. Sep. 2025].

[7] Introducing OpenAI o3 and o4-mini, September 2025.

[8] Kimi K2: Open Agentic Intelligence, September 2025. [Online; accessed 4. Sep. 2025].

[9] Anthropic. Claude opus 4.1, August 2025.

[10] Suma Bailis, Jane Friedhoff, and Feiyang Chen. Werewolf Arena: A Case Study in LLM Evaluation via Social Deduction. *arXiv*, July 2024.

[11] Noam Brown and Tuomas Sandholm. Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2017.

[12] Bill Chen and Jerrod Ankenman. *The Mathematics of Poker*. ConJelCo, LLC, 2006.

[13] Google. Kaggle game arena evaluates ai models through games. Google Technology AI Blog, 2025.

[14] Google. Release notes | gemini api | google ai for developers, April 2025.

[15] Leon Guertler, Bobby Cheng, Simon Yu, Bo Liu, Leshem Choshen, and Cheston Tan. TextArena. *arXiv*, April 2025.

[16] Ian Hendley. treys: A pure python poker hand evaluation library. `https://pypi.org/project/treys/`, 2017.

[17] Samuel Holt, Max Ruiz Luyten, and Mihaela van der Schaar. L2mac: Large language model automatic computer for extensive code generation. In *Proceedings of the Twelfth International Conference on Learning Representations*, 2024.

[18] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

[19] Michael Johanson. Measuring the size of large no-limit poker games. *arXiv preprint arXiv:1302.7008*, 2013.

[20] Bastian Koch, Latha R. Varshney, Florian Scheuerman, Florian Wenzel, Jacob Gal, Ben Shneiderman, Yulia Tsvetkov, Benjamin B. Bederson, John Saenz, Alejandro Cabrera, et al. Mapping global dynamics of benchmark creation and saturation in artificial intelligence. *Nature Communications*, 13(1):6793, 2022.

[21] Junnan Liu, Hongwei Liu, Linchen Xiao, Ziyi Wang, Kuikun Liu, Songyang Gao, Wenwei Zhang, Songyang Zhang, and Kai Chen. Are your llms capable of stable reasoning? *arXiv preprint arXiv:2412.13147*, 2024.

[22] Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V. Le, Barret Zoph, Jason Wei, and Adam Roberts. The flan collection: Designing data and methods for effective instruction tuning. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 22631–22648. PMLR, 23–29 Jul 2023.

[23] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisỳ, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

[24] James Simpson. pokersolver: Javascript poker hand solver. `https://github.com/goldfire/pokersolver`, 2016. Accessed: 2025-09-05.

[25] Team. Qwen3: Think Deeper, Act Faster. *Qwen*, April 2025.

[26] Design Arena Team. Design arena: Benchmarking llm-based visual design generation. arXiv preprint, 2024.

[27] Kernel Bench Team. Kernel bench: Evaluating llms on system-level programming tasks. Benchmark Repository, 2025.

[28] Ryan Teknium, Roger Jin, Jai Suphavadeeprasit, Dakota Mahan, Jeffrey Quesnelle, Joe Li, Chen Guang, Shannon Sands, and Karan Malhotra. Hermes 4 Technical Report. *arXiv*, August 2025.

[29] Alexander Wei, Sheryl Hsu, and Noam Brown. Alexander Wei on X: "1/N I'm excited to share that our latest @OpenAI experimental reasoning LLM has achieved a longstanding grand challenge in AI: gold medal-level performance on the world's most prestigious math competition—the International Math Olympiad (IMO).". *X* (formerly Twitter), September 2025. [Online; accessed 5 Sep. 2025].

[30] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Proceedings of the 37th International Conference on Neural Information Processing Systems*, volume 36 of *NeurIPS '23*, pages 11809–11822. Curran Associates, Inc., December 2023.

[31] Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, Sasha Cui, Sayash Kapoor, Shayne Longpre, Kevin Meng, Rebecca Weiss, et al. Establishing best practices for building rigorous agentic benchmarks. *arXiv preprint arXiv:2507.02825*, 2025.

[32] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In *Advances in Neural Information Processing Systems*, 2007.

A Additional quantitative results

A.1 Pairwise win rates

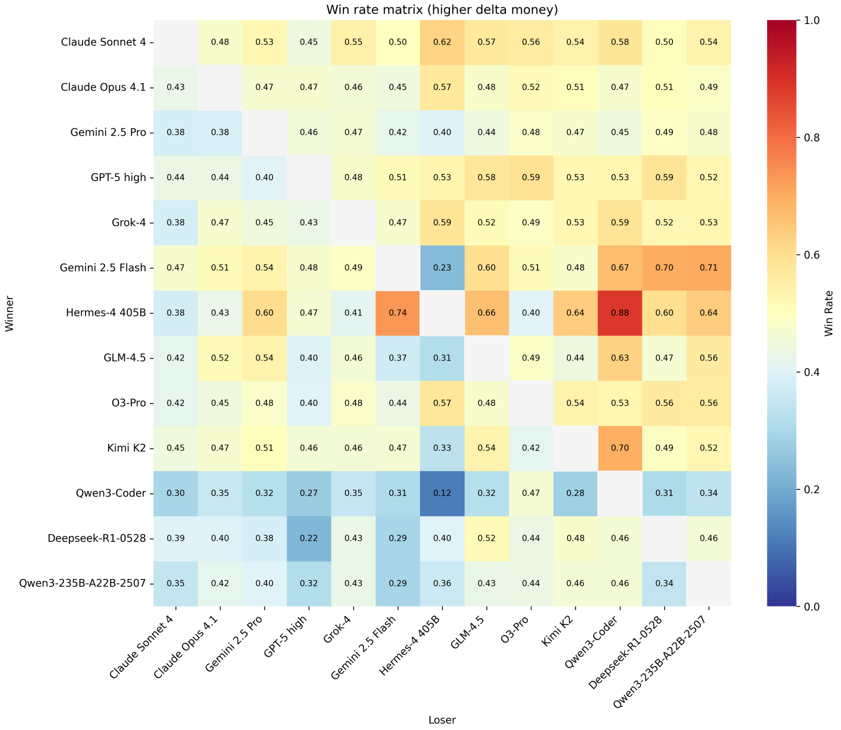

Figure 3: Pairwise win rate matrix. We say that a model A *wins* against another model B in a game if A's bot achieves a higher delta money than B's bot. Models never play against themselves so the main diagonal is omitted. Models ordered by increasing final rank.

A.2 Player performance outcomes

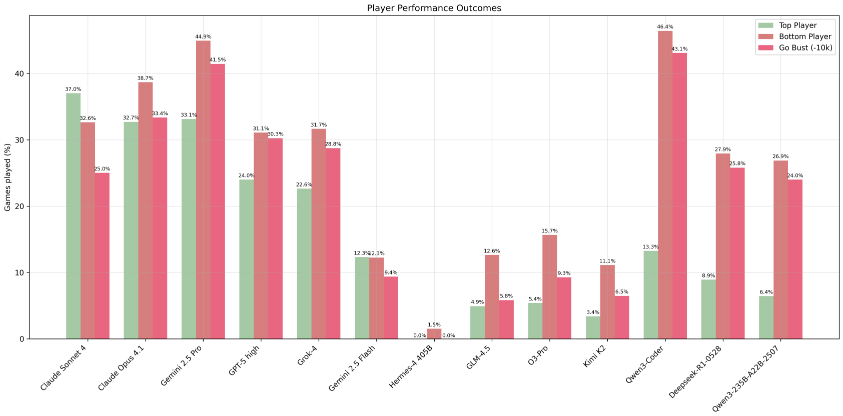

Figure 4: Player performance outcome frequencies for each model. We show the frequencies of achieving the highest delta money (top player), lowest delta money (bottom player), or delta money exactly -10000 (going bust). Models ordered by increasing final rank.

A.3 Delta money distribution

We show the delta money distribution by model in Figure 4. We find that most models exhibit fairly high variance across games, with the exception of Hermes-4 405B [28]. As noted in Section 3.2, Hermes-4 405B plays fairly defensively, usually folding.

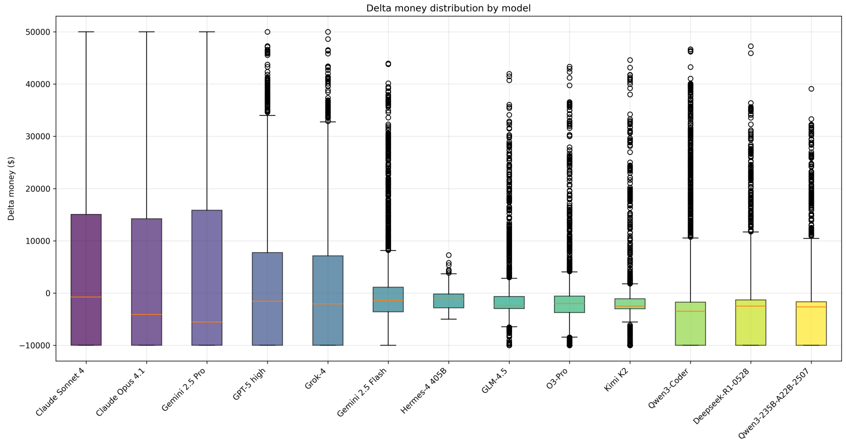

Figure 5: Delta money distribution by model. Models ordered by increasing final rank.

B Analysis of reasoning and bots by model

B.1 Gemini 2.5 Pro (Batch 5)

Gemini 2.5 Pro developed an overall aggressive poker bot that emphasized strategies to value bet with strong hands, calculate pot odds for draws, and perform selective bluffing. It uses tiered pre-flop hand rankings, detailed post-flop hand evaluation, and dynamic bet sizing to inform its strategy.

B.1.1 CoT analysis

The models began with developing the core logic where it first emphasized pre-flop hand strengths. Over the 5 iterations it improved on its strategy by incorporating robust hand ranking, pot odds, and aggressive value betting. The reasoning shows a step-by-step refinement.

Chain-of-Thought Excerpt

Model reasoning (excerpt):
"**Developing the Bot’s Core**
I’m structuring the poker bot as a ‘SimplePlayer‘ class in Python, inheriting from a base ‘Bot‘ class. The objective is clear: maximize ‘delta money‘ (ending chips minus starting chips) in No-Limit Texas Hold’em. This foundational framework feels solid, ready to be built upon. **Framing Hand Strength**
I’m now diving deep into hand strength evaluation. I’m planning to use a lookup table for pre-flop hand rankings. For post-flop, I’ll need a system to evaluate my hand against the community cards. Considering the time constraint, I might lean towards a simplified approach, perhaps calculating the raw poker hand rank (e.g., a pair beats a high card, etc.) and applying some weighting.
Prioritizing Core Functionality
I’ve clearly defined the critical components: hand strength evaluation, opponent modeling (though simplified initially), and a betting strategy based on those. I’m leaning towards a hand representation using numerical ranks and suits for efficiency. Pre-flop hand strength will rely on a lookup table or a ranking system to quickly assess starting hands. Post-flop hand assessment needs careful consideration to balance accuracy and speed.
Decomposing the Challenge
I’m now wrestling with the detailed components. Parsing card strings to usable rank/suit formats is a must. Hand strength is the core: pre-flop will use a ranking system; post-flop requires a hand ranker and equity calculations. I’m exploring a hand-ranking function and weighing Monte Carlo simulations against simpler öutsčalculations. I need to find the right balance between accuracy and speed.

Planning Player Actions
I'm now focusing on how the bot will make decisions during a round. My plan is to build a ‘SimplePlayer‘ class, inheriting from ‘Bot‘, that needs to maximize my ‘delta money‘, which is my final chips minus my starting chips. I must manage my hand strength, opponent modeling, betting strategies, and risk management.
Refining the Core Logic
The ‘SimplePlayer‘ class is taking shape. I'm confirming ‘on_start‘ receives my initial hand, and ‘on_round_start‘ provides the round state. This clarifies the game flow: ‘on_start‘ gets the initial cards, followed by ‘on_round_start‘ and ‘get_action‘ for each betting phase. This sets the stage for implementing hand ranking and betting decisions.
Mapping the Game Flow
I'm now interpreting the game rules' context. The crucial methods are clear: ‘on_start‘ gets dealt cards, and ‘on_round_start‘ provides the current betting state. Then, ‘get_action‘ is the central part of my gameplay decision. The focus is now on understanding how to build the core logic.
Evaluating Key Methods
I'm now working through the game lifecycle methods. I've confirmed ‘on_start‘ initializes the player and their hand, while ‘on_round_start‘ provides the current game state. This clears the stage for ‘get_action‘, the core of the betting logic. I'm figuring out how the data flows."
Mapping the Bot's Logic
I'm planning to break down the bot's decision-making process. I'll determine hand strength, consider my position, and model opponents (initially simplified). This information will be used to formulate betting strategies. The goal is simple: win and win consistently.
Refining the Game Logic
I'm now starting to think about the overall structure, and what each function should return. The critical piece is now the decision-making process in ‘get_action‘. Pre-flop decisions depend on hand ranking. Post-flop decisions involve combining hand strength with outs calculations. I'll start simple and build from there.
Evaluating Opponent Behavior
I'm now evaluating opponent actions for modeling. Analyzing their actions, particularly betting patterns, is a key. While detailed tracking is complex, I can start with VPIP and PFR as initial opponent indicators. This data will influence my betting decisions over time.
Implementing the Core Logic
I'm now refining the core components and hand evaluation functions. I'm working through the card parsing, the main hand evaluation function, pre-flop ranking lookup, and equity estimations using 'outs'. The ‘get_action‘ method is next, where everything needs to come together for betting decisions, based on the hand's strength and position at the table.
Iterating on the Strategy
I'm now tweaking the initial decision logic. The plan is to refine pre-flop hand rankings and thresholds for post-flop betting. I'm prioritizing a simpler, more robust approach for the first iteration, focusing on clear rules and effective execution. This approach aims for solid foundations.
Implementing the Strategy
I'm now starting to put the pieces together. I've defined the hand representation, which is now tuples. I'm coding the pre-flop hand ranking using a simplified categorized system, as this will be the first thing I do. The evaluation of the hands, and the main structure are next on my list.
Reviewing the Strategy
I'm now reviewing the overall strategy to ensure it aligns with the limited scope. I'm focusing on simplicity and robustness, aiming for a well-defined approach. I'll use a categorized approach for pre-flop and a simplified post-flop strategy based on hand ranking and outs.
Assessing the Hand Evaluation
I'm now moving forward with my hand evaluation function. I'm focusing on the algorithm for ‘evaluate_best_hand‘, making sure to efficiently rank hands. I'm planning to create a ranking based on the hand's category (e.g., straight flush, four of a kind) to speed up decision-making.
Focusing on the Strategy
I'm now streamlining the overall plan. I will use a categorical hand-ranking system to simplify pre-flop decisions. For post-flop, I am planning to use a simplified approach based on hand ranking and outs to make the betting strategies much easier to manage.

B.1.2 Code analysis

GLM 4.5's bot employs a tiered pre-flop strategy based on hand strength, a detailed hand evaluation system for post-flop play, and decision-making that incorporates pot odds, draws, and selective bluffing. It uses aggressive value betting for strong hands and disciplined folding for weak ones.

Pre-Flop Strategy (Tiered Hand Ranking)

```
# Hand strength tiers based on Sklansky-Malmuth rankings (simplified)
```

```python
# Using canonical representation: AKs, KQs, 77, T8o
self.tier1 = {'AA', 'KK', 'QQ', 'JJ', 'AKs'}
self.tier2 = {'TT', 'AQs', 'AJs', 'KQs', 'AKo'}
self.tier3 = {'99', 'JTs', 'QJs', 'KJs', 'ATs', 'AQo'}
self.tier4 = {'88', 'KTs', 'QTs', 'J9s', 'T9s', '98s', 'AJo', 'KQo'}
self.tier5 = {'77', '66', '55', 'A9s', 'A8s', 'A7s', 'A6s', 'A5s', 'A4s', 'A3s', 'A2s', 'K9s', 'KTo', 'QJo'}

def _get_preflop_strength(self, hand: List[str]) -> int:
    """Categorizes hand into a strength tier (1-6, 1 is strongest)."""
    card1, card2 = hand[0], hand[1]
    rank1, suit1 = card1[0], card1[1]
    rank2, suit2 = card2[0], card2[1]

    r1_val = RANKS.index(rank1)
    r2_val = RANKS.index(rank2)

    if r1_val < r2_val:
        rank1, rank2 = rank2, rank1

    is_suited = 's' if suit1 == suit2 else 'o'
    is_pair = (rank1 == rank2)

    hand_key = f"{rank1}{rank2}"
    if not is_pair:
        hand_key += is_suited

    if hand_key in self.tier1: return 1
    if hand_key in self.tier2: return 2
    if hand_key in self.tier3: return 3
    if hand_key in self.tier4: return 4
    if hand_key in self.tier5: return 5
    return 6
```

This code defines hand strength tiers for pre-flop decisions, categorizing starting hands from premium (tier 1) to marginal (tier 5) for strategic play.

Hand Evaluation (_evaluate_hand)

```python
def _evaluate_hand(self, hole_cards: List[str], community_cards: List[str]) -> Tuple[int, Tuple]:
    """Evaluates the best 5-card hand from 2 hole cards and community cards."""
    all_cards = self._parse_cards(hole_cards + community_cards)
    best_hand_rank = -1
    best_hand_kickers = ()

    if len(all_cards) < 5:
        return 0, tuple(sorted([c[0] for c in all_cards], reverse=True))

    for hand_combo in combinations(all_cards, 5):
        ranks, is_flush, is_straight, counts, rank_counts = self._get_hand_details(list(hand_combo))
        major_kickers = sorted([r for r, c in rank_counts.items() if c >= 2], reverse=True)
        kicker_ranks = sorted([r for r, c in rank_counts.items()], reverse=True)
        current_rank = 0
        current_kickers = tuple(kicker_ranks)

        if is_straight and is_flush:
            current_rank = 9 if ranks[0] == 14 else 8
        elif counts[0] == 4:
            current_rank = 7
            kicker = [r for r, c in rank_counts.items() if c == 1][0]
            current_kickers = (major_kickers[0], kicker)
        elif counts == [3, 2]:
            current_rank = 6
            current_kickers = (major_kickers[0], major_kickers[1])
        elif is_flush:
            current_rank = 5
        elif is_straight:
            current_rank = 4
        elif counts[0] == 3:
            current_rank = 3
        elif counts == [2, 2, 1]:
            current_rank = 2
        elif counts[0] == 2:
            current_rank = 1

        if current_rank > best_hand_rank:
            best_hand_rank = current_rank
            best_hand_kickers = current_kickers
        elif current_rank == best_hand_rank and current_kickers >
            best_hand_kickers:
            best_hand_kickers = current_kickers

    return best_hand_rank, best_hand_kickers
```

The hand evaluation function assesses the best possible 5-card combination, returning a rank (0-9) and kickers for strength comparison.

Post-Flop Action (_get_postflop_action)

```
def _get_postflop_action(self, round_state: RoundStateClient,
    remaining_chips: int, active_players: int, legal_actions: Set[
    PokerAction]) -> Tuple[PokerAction, int]:
    """Determines post-flop action based on hand strength and
        potential."""
    hand_power, _ = self._evaluate_hand(self.hand, round_state.
        community_cards)
    pot_odds = self._calculate_pot_odds(round_state)

    # Strong Hands (Two Pair or better)
    if hand_power >= 2:
        if PokerAction.RAISE in legal_actions:
            bet_amount = int(round_state.pot * random.uniform(0.6,
                0.9)) # Value bet
            bet_amount = min(round_state.max_raise, max(round_state.
                min_raise or 0, bet_amount))
            if bet_amount >= remaining_chips:
                return PokerAction.ALL_IN, remaining_chips
            return PokerAction.RAISE, bet_amount
        if PokerAction.ALL_IN in legal_actions: return PokerAction.
            ALL_IN, remaining_chips
        if PokerAction.CALL in legal_actions: return PokerAction.CALL,
            0
        return PokerAction.CHECK, 0

    # Medium Hands (One Pair)
    if hand_power == 1:
        if PokerAction.CHECK in legal_actions: return PokerAction.
            CHECK, 0
        amount_to_call = round_state.current_bet - round_state.
            player_bets.get(str(self.id), 0)
        if PokerAction.CALL in legal_actions and amount_to_call <
            remaining_chips * 0.2:
            return PokerAction.CALL, 0
        return PokerAction.FOLD, 0

    # Drawing hands / Bluffs
    outs = self._count_outs(self.hand, round_state.community_cards)
    if outs >= 8 and round_state.round != 'River': # Good draw (flush
       or open-ended straight)
        equity = self._calculate_equity_from_outs(outs, round_state.
           round)
        if pot_odds is not None and equity > pot_odds and PokerAction.
           CALL in legal_actions:
            return PokerAction.CALL, 0
        # Bluff if heads-up and everyone checked
        if active_players <= 2 and round_state.current_bet == 0 and
           PokerAction.RAISE in legal_actions:
            if random.random() < 0.3: # Bluff 30% of the time
                bluff_amount = int(round_state.pot * 0.5)
                bluff_amount = min(round_state.max_raise, max(
                   round_state.min_raise or 0, bluff_amount))
                if bluff_amount > 0: return PokerAction.RAISE,
                   bluff_amount
        if PokerAction.CHECK in legal_actions: return PokerAction.
           CHECK, 0
        return PokerAction.FOLD, 0

    if PokerAction.CHECK in legal_actions: return PokerAction.CHECK, 0
    return PokerAction.FOLD, 0
```

Post-flop decisions prioritize value betting strong hands, calling with medium strength if pot odds favor it, and pursuing draws or selective bluffs based on outs and opponents.

B.2 GLM 4.5 (Batch 5)

The GLM 4.5 model developed a defensive poker bot which focuses on tight play with simple heuristic-based hand evaluation. It emphasizes preflop aggression with strong hands and postflop decisions based on basic hand strength metrics. It has an overall less detailed and thought out strategy compared to Gemini.

B.2.1 CoT Analysis

The model's reasoning developed a simple bot with basic strategies for preflop and postflop action. The model considered using external libraries like `pokersolver` [24] or `treys` [16] but opted for a custom, simplified hand evaluator to keep the code short and pure Python. For pre-flop, it outlined a Chen-like formula [12] for hand value, assigning points based on ranks, suits, and connectivity. For postflop, it designed a rank-counting system to return a float strength value. Throughout its iterations, the model focused on fixing code errors it faced. It also refined its strategy, like adjusting the raise amounts and incorporating pot odds approximations without full Monte Carlo simulations to stay within time limits.

Chain-of-Thought Excerpt

Model reasoning (excerpt): "We'll design a simple bot that uses a basic strategy: Pre-flop: Play tight-aggressive. Only play with strong hands (e.g., pairs, high cards) and raise with premium hands. Post-flop: Bet when we have a strong hand (like top pair, two pair, etc.) and fold weak hands. [...] We'll implement a very simple bot that does not do hand evaluation but instead uses the hole cards and community cards in a naive way. [...] We'll implement a very short hand evaluator that only checks for pairs, two pair, three of a kind, and then uses the high card. It won't check for straight or flush to save space."

Chain-of-Thought Excerpt

Model reasoning (excerpt): "We'll do a very simplified version: points = max(card1_rank, card2_rank) [converted to a number] if cards are paired: points = points * 2 if suited: points += 2 if connected: points += 2 [...] Then we set thresholds: points >= 16: raise points >= 10: call else: fold [...] Post-flop: We calculate hand_strength(hole_cards, community_cards) and then compare it with a threshold. [...] Then: If hand_strength_value >= 3.0 (three of a kind or better), we raise or all-in. If hand_strength_value >= 1.0 (one pair or better) and we are facing a bet that is not too large, we call. Otherwise, fold."

Chain-of-Thought Excerpt

Model reasoning (excerpt): "We must be cautious: the hand representation: e.g., ['Ah', 'Ks'] for Ace of hearts and King of spades. [...] We'll split each card string into rank and suit. [...] We'll map the letter ranks to numbers: 'A' -> 14 'K' -> 13 'Q' -> 12 'J' -> 11 'T' -> 10 others: int(rank) [...] Now, in get_action: round_name = round_state.round # 'Preflop', 'Flop', 'Turn', 'River' if round_name == 'Preflop': points = self._preflop_hand_value(self.hole_cards) [...] my_bet = round_state.player_bets.get(str(self.id), 0) amount_to_call = max(0, current_bet - my_bet) [...] We hope that the game server calls on_start for every hand. If not, this bot will use the same hole cards for every hand, which is bad, but we have no choice."

Chain-of-Thought Excerpt

Model reasoning (excerpt): "Since we are on iteration 3, we are building upon the previous code. However, the previous code is not provided. We must assume that the previous code had issues and we are to fix them. [...] We'll implement the bot with the above ideas. We'll code: 1. A helper function for converting a card to a rank number and suit. 2. A helper function for pre-flop hand strength (Chen-like). 3. A helper function for post-flop hand strength (simplified). [...] We'll also avoid using any external library to keep the code short and requirements.txt empty. [...] We'll assume: the list player_hands has one element per player, and that element is a string of space separated two cards? [...] We'll do: self.hole_cards = my_hole_cards_str.split() [...] We'll be as concise as possible. We'll avoid long code."

B.2.2 Code Analysis

The final bot implements a conservative heuristic-driven poker strategy that relies on simple hand strength calculations. Decisions prioritize folding weak hands, calling moderate ones with favorable pot odds approximations, and raising or all-in with strong holdings.

Listing 1: Hand Strength Helper Functions

```python
def _rank_to_number(self, rank_char):
    if rank_char == 'A':
        return 14
    elif rank_char == 'K':
        return 13
    elif rank_char == 'Q':
        return 12
    elif rank_char == 'J':
        return 11
    elif rank_char == 'T':
        return 10
    else:
        return int(rank_char)

def _get_rank_suit(self, card_str):
    return card_str[0], card_str[1]

def _preflop_hand_value(self, hole_cards):
    if not hole_cards or len(hole_cards) < 2:
        return 0
    rank1, suit1 = self._get_rank_suit(hole_cards[0])
    rank2, suit2 = self._get_rank_suit(hole_cards[1])
    num1 = self._rank_to_number(rank1)
    num2 = self._rank_to_number(rank2)
    points = max(num1, num2)
    if num1 == num2:
        points = num1 * 2
    else:
        if suit1 == suit2:
            points += 2
        if abs(num1 - num2) == 1:
            points += 2
    return points

def _postflop_hand_strength(self, hole_cards, community_cards):
    all_cards = hole_cards + community_cards
    if not all_cards:
        return 0.0
    rank_count = {}
    for card in all_cards:
        rank, _ = self._get_rank_suit(card)
        num = self._rank_to_number(rank)
        rank_count[num] = rank_count.get(num, 0) + 1
    counts = sorted(rank_count.values(), reverse=True)
    if counts[0] == 4:
        return 7.0
    elif counts[0] == 3 and (len(counts) > 1 and counts[1] >= 2):
        return 6.0
    elif counts[0] == 3:
        return 3.0
    elif len(counts) > 1 and counts[0] == 2 and counts[1] == 2:
        return 2.0
    elif counts[0] == 2:
        return 1.0
    else:
        max_rank = max(rank_count.keys()) if rank_count else 0
        return min(max_rank / 14.0, 0.9)
```

Listing 2: Core Decision-Making in get_action

```
def get_action(self, round_state: RoundStateClient, remaining_chips:
   int) -> Tuple[PokerAction, int]:
    try:
        current_bet = round_state.current_bet
        min_raise = round_state.min_raise
        max_raise = round_state.max_raise
        pot = round_state.pot
        round_name = round_state.round
        community_cards = round_state.community_cards
        my_bet = round_state.player_bets.get(str(self.id), 0)
        amount_to_call = max(0, current_bet - my_bet)
        if round_name == 'Preflop':
            hand_value = self._preflop_hand_value(self.hole_cards)
        else:
            hand_value = self._postflop_hand_strength(self.hole_cards,
                 community_cards)
        if round_name == 'Preflop':
            if hand_value >= 16:
                raise_amount = min(max_raise, current_bet + min_raise
                    * 2)
                return (PokerAction.RAISE, raise_amount)
            elif hand_value >= 10:
                if amount_to_call == 0:
                    return (PokerAction.CHECK, 0)
                elif amount_to_call <= pot // 4:
                    return (PokerAction.CALL, 0)
                else:
                    return (PokerAction.FOLD, 0)
            else:
                return (PokerAction.FOLD, 0)
        else:
            if hand_value >= 6.0:
                return (PokerAction.ALL_IN, 0)
            elif hand_value >= 3.0:
                raise_amount = min(max_raise, current_bet + pot // 2)
                return (PokerAction.RAISE, raise_amount)
            elif hand_value >= 1.0:
```

```python
                if amount_to_call == 0:
                    return (PokerAction.CHECK, 0)
                elif amount_to_call <= pot // 5:
                    return (PokerAction.CALL, 0)
                else:
                    return (PokerAction.FOLD, 0)
            else:
                if amount_to_call == 0:
                    return (PokerAction.CHECK, 0)
                else:
                    return (PokerAction.FOLD, 0)
    except Exception:
        return (PokerAction.FOLD, 0)
```

B.3 DeepSeek-R1-0528 (Batch 5)

DeepSeek-R1-0528 developed a poker bot that focused on optimizing preflop decisions using simplified hand strength calculations and postflop play based on a custom hand evaluator. It aimed to balance aggression with caution, incorporating pot odds and basic bluffing.

B.4 CoT analysis

The model thought through and developed a simplified hand evaluator with quick preflop ranking formulas and limited Monte Carlo iterations for postflop strength estimation.

Chain-of-Thought Excerpt

Model reasoning (excerpt): "Let's design a bot that is competitive but with concise code. Strategy: We will implement a bot that uses a simplified version of the 'TAG' (tight-aggressive) strategy, with adjustments for position and hand strength."

The model refined its approach by incorporating the Chen-like formula [12] for preflop evaluation and a full 7-card hand evaluator for postflop. It also implemented bluffing on scare cards to address passive play.

Chain-of-Thought Excerpt

Model reasoning (excerpt): "We are given 10 games of heads-up (two players) in the feedback. Our bot lost every game, with an average loss of 7.5 chips per game. Key observations: The bot lost 10 out of 10 games, which indicates the strategy is too passive or not aggressive enough in stealing blinds, or too loose when behind. Proposed improvements: 1. Preflop strategy: Use a predefined range of starting hands to decide whether to fold, call, or raise. 2. Postflop: Evaluate hand strength using approximate methods but in a simplified way to keep code short. We'll use a preflop hand chart and then for postflop, we'll use a simple hand evaluator that returns a rank (from 0 to 9) for the hand."

Throughout the iterations Deepseek kept optimizing the code. It reduced the Monte Carlo samples to 10, and focused on adding valid action checking.

Chain-of-Thought Excerpt

Model reasoning (excerpt): "I've implemented a highly optimized poker bot with concise code while maintaining competitive strategy. The key improvements include: 1. Significantly condensed all functions, especially the hand evaluator, using efficient algorithms 2. Simplified strategy implementation while retaining core aspects: - Preflop uses quick hand ranking formula instead of large lookup tables - Postflop evaluates hand strength with monte carlo estimation (10 iterations for speed) - Implements pot odds and position-aware betting - Aggressive play for strong hands, cautious for weak ones 3. Added proper error checking and handling for all edge cases 4. Optimized memory usage by minimizing stored data 5. Handled all possible game states including side pots and all-ins"

B.4.1 Code analysis

The final bot uses a custom hand evaluator for 5-7 card combinations, calculating strength by simulating opponent hands and comparing scores. The strategy can be described as tight-aggressive, with position-aware decisions and pot odds based decisions.

Hand Strength Calculation

```python
def _get_action_strength(self, hand, community):
    samples = 10 if len(community) < 5 else 1
    wins = 0
    our_hand = hand + community
    max_score = self._eval_hand(our_hand)
    all_cards = [f"{rank}{suit}" for rank in "23456789TJQKA" for suit
        in "shdc"]
    for card in our_hand:
        if card in all_cards:
            all_cards.remove(card)
    for _ in range(samples):
        random.shuffle(all_cards)
        opp_hand = all_cards[:2]
        opp_score = self._eval_hand(opp_hand + community)
        if max_score > opp_score:
            wins += 1
    return wins / max(samples, 1)
```

Action Decision Logic

```python
def get_action(self, round_state: RoundStateClient, remaining_chips:
    int) -> Tuple[PokerAction, int]:
    player_id_str = str(self.id)
    player_bet = round_state.player_bets.get(player_id_str, 0)
    call
```