

Optimising Team Compositions through Large Language Model Theorcrafting

1st George E.M. Long
Queen Mary University of London
g.e.m.long@qmul.ac.uk

2nd Diego Perez-Liebana
Queen Mary University of London
diego.perez@qmul.ac.uk

3rd Spyridon Samothrakis
University of Essex
ssamot@essex.ac.uk

Abstract—The process of exploring and analysing the rules of the game to develop optimal strategies can be known as Theorcrafting. Sometimes, when some strategies dominate over others, designers may require the game rules to be changed in order to account for the dominance of these strategies (a process called Rebalancing). We propose a game-agnostic method which uses agentic Large Language Models to engage in theorcrafting, allowing for these optimal strategies to be discovered and accounted for by game designers while the game is still in development. Our results show that the LLMs are able to utilise their theory to beat each experiment’s benchmark, while requiring fewer games to be played compared to Evolutionary Algorithms. Additionally, some emergent behaviour was discovered which could potentially improve performance further.

Index Terms—LLM, Large Language Model, Theorcrafting, Game Balance, Agentic AI, Game AI

I. INTRODUCTION

Competitive games, some of the most popular worldwide, involve competing against other players to win. These games take many forms, whether it’s Chess, Football, or Deadlock. However, in order to win, some players take this competition to a further dimension, the ‘metagame’. Within competitive games, the metagame refers to the collection of strategies which are perceived to be the best [1].

Game balancing, although it lacks a concrete definition, can refer to the process where designers modify the rules of their game to achieve a certain goal [2]. Within competitive games, these goals are usually to allow for a wide variety of strategies to be employed, while ensuring that player skill impacts victory chance. However, the metagame can cause friction with these goals: if the metagame focuses on a small subset of dominant strategies, a situation can arise where employing a dominant strategy becomes the key to victory, reducing the impact of player skill, and resulting in a smaller variety of viable strategies. To combat this, game designers will add new content, and balance existing mechanics to disrupt the metagame, resulting in the start of the Ignorance stage of the metagame cycle [1].

Within this Ignorance stage, the process of theorcrafting starts to emerge. Theorcrafting, popularised in MMORPGs such as World of Warcraft, can be seen as the act of exploring

and analysing the rules and mechanics of a game to create and improve strategies within the game [3].

Theorcrafting, while an enjoyable practice for those who engage in it, can have a fundamental impact on how a game is played [4]. Additionally, this impact can increase elitism in the player base, as players are pressured to adopt theorcrafted strategies, possibly reducing enjoyment of the game for casual players [3].

One important aspect of theorcrafting involves parsing the rules of a game to develop theories on how to best play the game. This task of natural language processing is well-suited to Large Language Models (LLMs). LLMs have seen increasing use in Games Research [5], and there has been research on topics related to theorcrafting, such as using LLMs to draft cards in Magic: The Gathering [6], or as game-playing agents to aid in balancing [7]. However, to our knowledge, there has been no work on using Large Language Models specifically for the purpose of theorcrafting.

There has also been interest in interfacing Large Language Models with external systems. One approach to accomplish this is by giving the LLM ‘tools’ it is able to use to interact with other applications and systems. These tools act as functions the LLM is able to call, optionally with parameters to augment the logic of the tool. The Model Context Protocol (MCP) is an open-source standard for achieving this interaction [8]. It was developed by Anthropic, who open-sourced it in 2024 [9]. The protocol defines a standard way clients (e.g. Claude) are able to interact with MCP Servers, which provide tools for interacting with external systems.

We propose a game-agnostic approach which uses Large Language Models to engage in theorcrafting. Our motivation for this is to advance towards a system where LLMs are able to identify dominant strategies in games that are in development, and provide a report on how their theories were developed. This could then be used by game designers to preemptively change the rules of their game to reduce the impact of these strategies, potentially reducing the negative effects of theorcrafting, and improving the balance of the game. It is important to note that we believe that theorcrafting can be a core part of the game experience for some players, and we would not want to eliminate it completely, but reduce its impact on other types of players.

Our approach utilises the Model Context Protocol to facilitate communication between the Large Language Model and a

game we created called Autosim. We create three LLM agents, each handling respectively the generation of units, creation of benchmarks for testing, and the theorycrafting process. The theorycrafting agent uses the theory it developed to create optimal armies for Autosim. We then compare how well these armies do compared to armies created by Evolutionary Algorithms, finding that the LLM develops theories which create stronger armies while requiring fewer game simulations. Additionally, we observe some emergent behaviour from the LLMs, including creating scripts to aid in theorycrafting, as well as exploiting the MCP server to gain an advantage.

II. BACKGROUND

The use of Artificial Intelligence to aid designers in game balancing is a topic of interest within the Game AI community. A common approach has been to use Evolutionary Algorithms (EAs) to balance game parameters. Mahlmann et al. employed EAs to balance Dominion [10]. Morosan and Poli used EAs in Ms Pacman and Starcraft, using a multi-objective approach which attempted to balance the game while minimising how many changes to the game were made [11]. Beyer et al. [12] designed an approach which mixes both manual balancing and evolutionary algorithms. This was then expanded by Preuss et al. who tested it on openRA, an open-source game based on Red Alert [13].

One issue with Evolutionary Algorithms is their sample efficiency. Games can be expensive both in time and memory to simulate, therefore methods to reduce the simulations needed have been explored. Volz et al. mention that a surrogate model may be required for EAs due to the cost of simulating [14]. Morosan and Poli furthered their work on Ms Pacman [11] by creating a neural network to predict fitness values, reducing simulations required [15].

There has been work on using AI to both balance and exploit the metagame. Long et al. proposed using automatic point cost estimation to identify overpowered units in wargames [16]. Crane et al. created two algorithms which focus on creating teams that counter the current metagame [17]. Reis et al. propose an adversarial approach, where balanced game parameters are created by one agent, while another agent attempts to exploit any gaps it can find [18]. Pendurkar et al. propose a bi-level optimisation solution where they treat exploitation as an inner level optimisation problem, which is then used to find a solution for the outer level problem (Meta Balance) [19].

Large Language Models have seen use in game balancing, and areas similar to Theorycrafting. Bertram created UrzaGPT, an LLM fine-tuned with domain knowledge for drafting decks in Magic: The Gathering [6]. Zeng et al. developed Rulesmith, a framework for balancing games which uses Large Language Models as Player agents for the strategy game CivMini [7]. To play CivMini effectively, the LLM agents are given a textual rulebook for the game. Park et al. created Orak, a multi-game framework to aid in both evaluating and training LLM Agents [20]. They used MCP to enable the interaction between the LLM Agents and the games.

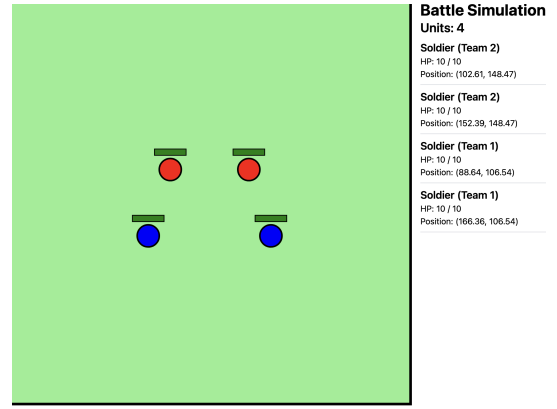


Fig. 1. Autosim as a web app. The green field is the map being played, and each team has 2 units. The sidebar contains the status of the units in the game.

Our work is, to the best of our knowledge, the first attempt at using LLMs to theorycraft optimal strategies in drafting-based games. We take motivation from previous work on increasing the sample efficiency of balancing algorithms, by showing this approach requires fewer games played than more traditional methods such as Evolutionary Algorithms. Additionally, LLMs provide a text-based explanation of the model's reasoning, allowing for insights into what game mechanics are exploited to create these optimal strategies.

III. METHOD

This section describes the method we developed to investigate the ability of Large Language Models (LLMs) to engage in theorycrafting. We first introduce the game created to perform these experiments, before describing how the LLM Agents are configured for the various tasks, and finally outline the setups for our experimental work.

A. Autosim

Large Language Models acquire knowledge through their training processes. As such, there is a risk that strategies that appear to be theorycrafted by the LLM are instead regurgitated strategies from its training data. To reduce the risk of this happening, we created a bespoke game for this paper: Autosim.

Autosim is a simple game inspired by Autobattlers such as Teamfight Tactics [21]. The game is built in Rust, with bindings for both Python and JavaScript. Python is used for headless runs for experiments, while JavaScript allows the game to be played on a website. Figure 1 shows an example of this game.

In Autosim, players create an army consisting of units they can buy using a budget. These armies then autonomously fight until one side has no remaining units. The game is fully deterministic and comprises three phases: drafting, placement and battle.

1) *Draft Phase*: In the draft phase, players have a fixed budget, which they can use to buy units to include in their army. Each unit has the following attributes which influence its performance in battle.

- Health - How much damage the unit can receive before it dies.
- Attack Strength - How much damage the unit's attacks do.
- Attack Speed - How many attacks the unit can make in 1 second.
- Attack Range - How far away the unit can attack.
- Defence - How much damage is reduced from attacks received.
- Speed - How fast the unit moves on the map.
- Cost - How much it costs to field this unit.

Players must try to find units which work well together to maximise performance of the army. Units are not hardcoded into Autosim; instead, units are defined via their properties in a JSON file, which will be parsed by the game to configure the units that will be used to play.

2) *Placement Phase*: In the placement phase, players must choose where to place the units they have chosen on the battlefield. The battlefield map is a 256×256 grid. Units move using a continuous system of coordinates on the map. The map is divided into a placement grid of 8×8 squares, in which players can place units in a 3×8 subgrid on their side of the map. Each cell in the grid can have one unit, which creates a maximum of 24 units per team.

3) *Battle Phase*: This is the decisive phase of the game. It takes place in real time, with 10 updates per second. When the battle starts, each unit will target the opponent closest to them. Units move toward their target until they are within attack range. Once within this range, they will begin charging up an attack, with the time needed to charge being based on the attack speed. If the unit they are targeting dies before the attack is fully charged up, the attack will reset and have to charge up again to attack a new target (also selected using the same criteria: the closest opponent).

Once the attack has been fully charged up, the unit will attack the target using the following equation to calculate the damage done: $\max(1, A - D)$, where A is the unit's attack strength, and D is the target's defence. Each attack will do at least 1 damage, preventing stalemates. Attacks eliminate their target if their health is reduced to 0, leading the attacking unit to select a new target. This cycle of units finding targets, moving to attack them, and dealing damage will repeat until one team has no units left on the field, in which case, the team with units left standing will be victorious.

B. LLM Agents

In our experiments, the LLMs perform multiple tasks. We chose to delegate these tasks to three separate agents, which we analyse here. We used both OpenAI's Codex and Anthropic's Claude desktop apps to host the agents. We used a variety of models, which will be specified in the experimental setup.

A base MCP Server was used as the basis for specific servers for the Benchmarking and Theorcrafting agents. This contains common tools which both agents will need to use. They are as follows:

- Reset: Resets the state of the MCP server, used when the agent starts a new session.
- Set Budget: Sets the budget each army has to buy units.
- Set Unitset: Sets the units to be used for the game.
- Set Tag: Sets a tag to be used for categorising runs.
- Add Unit: Adds a unit to the LLM's army at the chosen position.
- Remove Unit: Removes a unit from the army.
- Clear Army: Clears the army of all units.
- View Army: Returns a textual description of the current army composition.

The following subsections will go into the purpose of each agent and how they are set up. All source code, instruction files and prompts can be found in the link provided in the footnote on the first page.

1) *Unit Generation*: As mentioned in the previous section, Autosim does not come with any included units, thus units will have to be generated. We delegate to one LLM agent to create the collection of units for the game, which we term *Unitset*. The agent has access to a markdown document containing the rules of Autosim (rules.md), as well as a second file (AGENT.md for Codex, CLAUDE.md for Claude) that contains the instructions for the LLM agents. Notably, the following points were included in the LLM agent instructions:

- Create units based on the theme provided by the user.
- Ensure a variety of units with different costs and stats.
- Ensure the units are balanced.
- Assume that a standard army will have a budget of 50.
- Assume that a standard army will have around 20 units.

These were included to help shape what the generated unitset will look like, and to provide a minimum level of balance. The units are generated as JSON which can then easily be passed to Autosim. To interact with the agent, a variation on the following prompt is used:

Create a *THEME* unitset comprising of N units.

To ensure the Unitsets created by the LLM were reasonable, two tests were performed on them:

- 1) 100,000 games were played where both armies were randomly generated. This was to test the balance of the unitset.
- 2) 100,000 games were played where a randomly generated army would play against itself. This was to test the First Player Advantage of the Unitset.

In order to be considered acceptable, we used the criterion that the win rates of the first test should be close to a 50-50 ($\pm 5\%$). For the second test the first player win rate should be between 40% and 60%.

2) *Benchmark Armies*: We also need a collection of armies strong enough to act as a good benchmark against armies created through theorycrafting. To accomplish this, we use an agent whose purpose is to create a benchmark list of armies. The agent has access to the rules of the game formatted in markdown, as well as the unitsets created by the previous LLM agent. To perform this task, the agent has access to the Autosim Teambuild MCP server. This provides, in addition to the common tools mentioned previously (see Section III-B), the following functionality:

- Set No. Armies: Sets the number of armies to be created for the benchmark.
- Test Army: Tests the current army to ensure it is strong enough for the benchmark.
- Save Army: Saves a successfully tested army to the benchmark.
- Finish Teambuild: Finishes the team-building process for the benchmark, saves the created benchmark, and asks the LLM for a summary to be provided as markdown, providing an overview of the entire process.

To verify that the armies created by the agent are of an appropriate strength, they are played against 10,000 randomly generated armies. Each benchmark army created by this agent must win against at least 95% of these random teams. The MCP server will only allow armies to be saved if they meet this condition.

As with the previous agent, rules and agent instructions were provided, as well as the Unitset in JSON. To interact with this agent, a variation of this prompt is used:

Build 10 *THEME* armies, each with a budget of B .
Use the *TAG* tag.

The teams created by this agent will be saved as a benchmark under a specific name; the tag is used to categorise the runs. This will be used by the following agent to evaluate the army they have theorycrafted.

3) *Theorycrafting*: Finally, we need an agent to be able to perform theorycrafting on Autosim. The agent outlined here performs this task, using the Autosim Evaluate MCP server. This is used by the agent to theorycraft teams which are able to beat the benchmark. It has the common tools mentioned previously, as well as these additions:

- Set Benchmark: The benchmark to be evaluated against.
- Evaluate Army: Evaluates the LLM’s army against all armies in the benchmark.
- Fight Opponent: Plays against one benchmark army with detailed results being returned.
- Finish Theorycrafting: Finishes the theorycrafting process, saving results such as which armies were created by the agent, how well they did, and how many games were played while theorycrafting.

Along with the MCP server, the agent has access to the rules, the unitset to be used, and the agent instructions. The

instructions are broken up into three parts: initial theory, evaluation and final discussion.

The first part outlines how to come up with an initial theory and how to translate this theory into an army for the game. We instruct the LLM to first read the rules of the game, as well as the Unitset JSON, and then outline its understanding of the game rules. It should then identify exploitable mechanics, and which units are most able to take advantage of them. Finally, we tell the LLM to create a theory based on this information, and create an army using it.

The second part involves improving the current working theory of the LLM. The agent can use the Evaluate Army tool to see how many of the benchmark opponents are beaten. For this, two games are played against each opponent, once as player 1 and once as player 2. The agent’s aim is to win every game against every opponent. If they are unable to achieve this, they can use the Fight Opponent tool, which allows them to play a game against a specific opponent in the benchmark, getting detailed statistics about the gameplay. This includes a transcript of the battle, showing which units were targeted, when attacks were made, and when units died. Finally, a summary of each unit’s performance is provided, giving information such as how many opponents a unit killed, how much damage they dealt and received, and whether they survived until the end.

Finally, the third part in the instructions details how to save the results of the theorycrafting session, as well as producing an overview of the session and what has been learnt about the game through theorycrafting, to be given to the user.

Unlike the other two agents, there are two separate sets of instructions. This is because of emergent behaviour discovered while experimenting. While theorycrafting, the agent created scripts to aid the theorycrafting process. Although this may not be a problem in itself, one of these scripts was an exhaustive search on the problem space, which brute-forced the solution and created an unfair advantage for the LLM over the Evolutionary Algorithms. Because of this, we decided to create one agent which is allowed to script (but only play games through the MCP server, to prevent it from exhaustive searching again), and one agent which is prohibited from creating scripts. Our aim is to observe what impact the ability to script has on agent performance. Each prompt is a variation of the following where the final sentence was added for the non-scripting agent:

With a budget of B , theorycraft an army which is able to beat the *BENCHMARK* benchmark using the *UNITSET* unitset. Use the *TAG* tag. [DO NOT DO ANY SCRIPTING.]

If an agent took over an hour to come up with a response, we steered them towards finishing the task by telling them to do one more evaluation, and then call the Finish Theorycrafting tool.

C. Experimental Setup

We carried out experiments to measure the ability of Large Language Models to theorycraft. To act as a comparison to the Large Language Models, we created an Evolutionary Algorithm implemented in the DEAP Python library. Three variations of the algorithm were made, each with a different population size (10, 100, or 1000).

The genotype is a list of 24 indexes, one for each position a unit can start on. Each index can be an integer value from 0 to N , where N is how many units are in the chosen Unitset. Each value from 0 to $(N - 1)$ represents a unit in the unitset, while a value of N represents no unit being at that position.

Tournament Selection was used to select individuals to reproduce, with a tournament size of 5. One-Point Crossover was used to perform crossover on individuals, and Uniform Mutation was used to mutate individuals, with the range of values a gene can mutate to being 0 to N , and the probability of a gene mutating being 5%. 10% of the population was reserved for elites on each generation. If after crossover and mutation, an individual breaks the constraints of the army (budget or unit count), a repair function is used. This function first checks to see if there are any units with a count higher than the limit. If so, it will randomly remove units of that type from the army until the limit is no longer breached, it will then remove any units that make the army over budget, prioritising removing units that will make the army exactly meet the budget. The algorithm will run for 50 generations, with a population size depending on the instance. A limit of 50 calls to the Evaluate Army tool was imposed on the LLMs to match the number of generations.

All our experiments share the same budget for both the Theorycrafting and Benchmarking agents (50) and number of armies in the quality benchmark (10). Both LLMs and EAs had 5 runs for each of the following experiments:

1) *Fantasy Experiment*: This experiment is designed to test how well the LLM is able to theorycraft in a standard scenario. It uses a Unitset created by the generation agent powered by the GPT-5.3-Codex model with reasoning set to *High*. The following LLM-Models were tested on this experiment: GPT-5.3-Codex (Scripting and Non-Scripting), Claude Sonnet 4.6, and Haiku 4.5. Experiments were also conducted with GPT-5.1-Codex-Mini; however, it was not able to follow the instructions consistently for the experiments, and thus, it was not included in the results.

2) *Sci-Fi Experiment*: Like the previous experiment, this one is designed to see how well an LLM can theorycraft. The key difference is that the Unitset and benchmarks for this experiment were created by Claude Sonnet 4.6. This aims to test whether the choice of model used in creation of the scenario can affect the difficulty of theorycrafting powerful armies.

3) *Fantasy Switched Experiment*: This experiment aims to measure if there is any impact on using different models for the tasks of generating the Unitset and benchmark. Here we use GPT-5.3-Codex to create the Unitset, and Claude Sonnet 4.6 to build the benchmark armies.

4) *Sci-Fi Switched Experiment*: The final experiment is similar to the previous one, with the models swapped: we use Claude Sonnet 4.6 to create the Unitset, and GPT-5.3-Codex to build the benchmark of armies.

IV. RESULTS

A. Unitsets and Army Benchmarks

Two Unitsets were created for the experiments. The first, 'Fantasy', was created by GPT-5.3-Codex. It contained 50 units, with unit costs ranging from 1 to 9. Notably, the agent said "I used a fantasy roster with clear roles: swarm, frontline, skirmisher, ranged, assassin, siege, and late-game elites", while making sure that "Higher-cost units are stronger but include tradeoffs (speed, range, attack speed, or survivability profile)".

As previously mentioned, each Unitset needed to pass two tests to be considered for use in the experiments. This Unitset passed both tests, achieving a 50.60% - 49.40% matchup in the random test, and a first player advantage of 58.88% in the second test, both within the required thresholds. Each Unitset was successfully created in a single prompt.

The Sci-Fi Unitset was created by Sonnet 4.6. It contained 50 units, with these units having a cost from 1 to 12. It passed the first test, receiving results of 49.59%-50.41%, and passing the second test with a first player advantage of 56.73%.

An army benchmark was created for each experiment, where each army in the benchmark needed to defeat at least 95% of randomly generated armies, to ensure robustness. These were all created in one prompt, albeit in some cases a minimal number of iterations was needed for certain armies to pass the test. The agents were asked to provide a summary of the benchmark creation process, giving insight into how they created them. As an example, the summary for the Sci-Fi benchmark constructed by Sonnet 4.6 mentions that the "Annihilator Prime" unit was the "MVP" of the Unitset, having the "highest DPS/cost", while being "durable enough to sit at the frontline". Meanwhile, the summary from the GPT-5.3-Codex benchmark echoes this, saying "Top-end power is concentrated in highly stat-efficient late-cost units" such as the "Annihilator Prime" or "Ion Dreadnought".

B. Optimising Team Compositions

Figure 2 shows the results for the Fantasy Experiment. The Y axis represents the win rate against the benchmark, while the X axis is how many games were played at the point of evaluation. Note that the X axis is in logarithmic scale. Each LLM and EA had 5 runs, with each run being plotted.

We can see that two groups have formed. The Large Language Models have broadly managed to beat the benchmark while playing a much lower number of games compared to the evolutionary algorithms. Figure 3 shows only the LLM runs (with a linear scale), and from this we can see that Haiku 4.5 performed considerably worse than any of the other LLMs, with its best run managing to get a win rate of only 75%, and with some runs getting a win rate of 0%. We attribute this to the smaller model size compared to the other LLMs tested, which may impact its ability to generalise the problem.

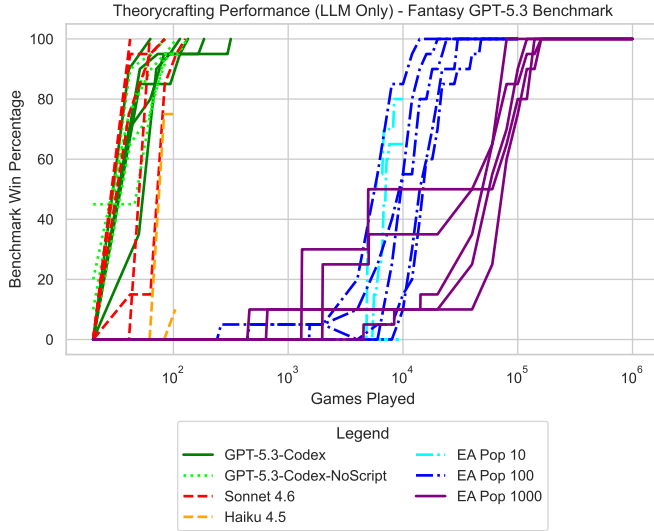


Fig. 2. The results of the fantasy experiment (GPT Unitset & benchmark). The X axis is the number of games played (logarithmic scale). The Y axis is the percentage of benchmark opponents defeated.

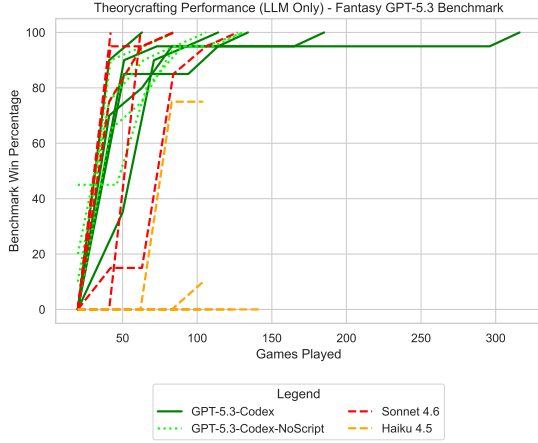


Fig. 3. The results of the fantasy experiment (GPT Unitset & benchmark) with only LLM agents. The X axis is the number of games played (linear scale). The Y axis is the percentage of benchmark opponents defeated.

The other LLMs (GPT-5.3-Codex, GPT-5.3-Codex-Noscript and Sonnet 4.6) performed similarly, with the exception of two GPT-5.3-Codex runs that had to play more games.

The Evolutionary Algorithms with a population size of 100 and 1000 managed to get a win rate of 100%, while the population size of 10 did not. This suggests that the problem space was large enough that the smaller population size was not able to explore it properly. Despite the fact that the larger population sizes managed to beat the benchmark, they did this while playing many more games than the LLMs.

Figure 4 and Figure 5 are heatmaps showing unit presence in all of the armies created by evolutionary algorithms and large language models respectively. The X axis of the heatmap is a discrete list of all the units in the fantasy Unitset, while the Y axis is the win rate of an army, resulting in each square in the

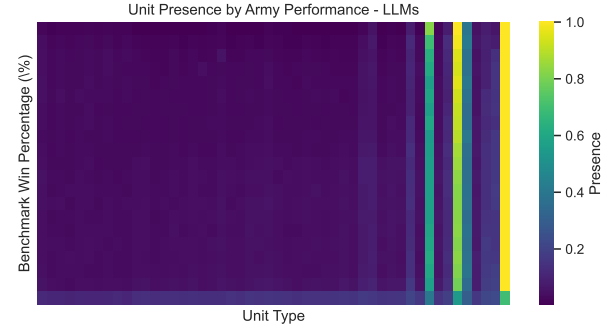


Fig. 4. Unit presence within all EA armies. X is the unit type, Y is the win rate, and each tile shows how many armies had at least one unit of that type.

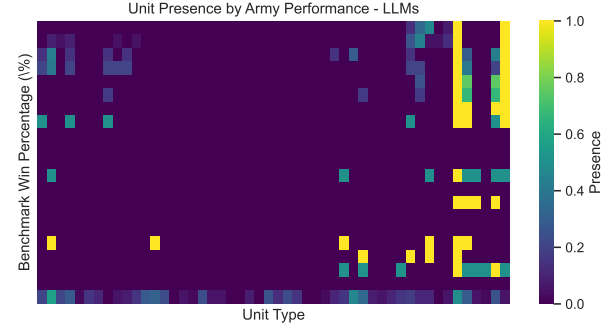


Fig. 5. Unit presence within all LLM armies. X is the unit type, Y is the win rate, and each tile shows how many armies had at least one unit of that type.

heatmap showing, for all the armies which achieved a certain win rate, how many contained at least one copy of the unit from the X axis. This allows us to see that both algorithms favoured two units, the "Archmage Supreme" and the "Titan of Dawn", with all of the armies that got 100% against the baseline including at least one of each. We can also see there is a large number of units with no presence except within armies which score 0% against the benchmark. This suggests that the Unitset is unbalanced, with the presence of only 2 of 50 units determining the game's outcome.

As mentioned previously, the agent provides a summary of the theorycrafting process. Looking at the summaries for this experiment, one of Sonnet 4.6's summaries mentions that "the Archmage Supreme + Titan of Dawn are the best units" and that "together they are unbeatable", while one of GPT-5.3-Codex-Noscript's entries says that the "Triple Titan frontline survives long enough for triple Archmage backline to win DPS race". These summaries, along with Figure 5, suggest that the LLMs identified the game's dominant strategy.

C. Theorycrafting Efficiency

Tables I, II, III, and IV show the efficacy (average win rate, higher is better) and efficiency (games needed to reach top performance, lower is better) of each model tested. The best performing model is highlighted in bold in each table. From these we can see that GPT-5.3-Codex (in both of its variants) was the most successful LLM, with every run managing to

Fantasy	Avg. win rate	Avg. Games to Max win rate
GPT	100% $\pm$ 0.00	162 $\pm$ 43.09
GPT No Scripts	100% $\pm$ 0.00	108 $\pm$ 10.29
Sonnet	100% $\pm$ 0.00	80 $\pm$ 14.16
Haiku	17% $\pm$ 14.63	49 $\pm$ 18.31
EA - Pop. 10	29% $\pm$ 17.92	3212 $\pm$ 1958.78
EA - Pop. 100	100% $\pm$ 0.00	27200 $\pm$ 5817.22
EA - Pop. 1000	100% $\pm$ 0.00	132000 $\pm$ 14966.63

TABLE I

RESULTS FOR THE FANTASY (GPT) UNITSET AND GPT BENCHMARK.

Fantasy Switched	Avg. win rate	Avg. Games to Max win rate
GPT	100% $\pm$ 0.00	33 $\pm$ 13.40
GPT No Scripts	100% $\pm$ 0.00	132 $\pm$ 77.59
Sonnet	99% $\pm$ 1.00	67 $\pm$ 15.13
Haiku	96% $\pm$ 2.92	86 $\pm$ 32.77
EA - Pop. 10	100% $\pm$ 0.00	3360 $\pm$ 1047.66
EA - Pop. 100	100% $\pm$ 0.00	6000 $\pm$ 632.46
EA - Pop. 1000	100% $\pm$ 0.00	25396 $\pm$ 8955.49

TABLE II

RESULTS FOR THE FANTASY (GPT) UNITSET AND SONNET BENCHMARK.

Sci-Fi	Avg. win rate	Avg. Games to Max win rate
GPT	100% $\pm$ 0.00	67 $\pm$ 26.16
GPT No Scripts	100% $\pm$ 0.00	35 $\pm$ 6.16
Sonnet	100% $\pm$ 0.00	55 $\pm$ 5.03
Haiku	98% $\pm$ 1.22	92 $\pm$ 8.32
EA - Pop. 10	100% $\pm$ 0.00	1760 $\pm$ 365.51
EA - Pop. 100	100% $\pm$ 0.00	2788 $\pm$ 783.11
EA - Pop. 1000	100% $\pm$ 0.00	4236 $\pm$ 2133.75

TABLE III

RESULTS FOR THE SCI-FI (SONNET) UNITSET AND SONNET BENCHMARK.

Sci-Fi Switched	Avg. win rate	Avg. Games to Max win rate
GPT	100% $\pm$ 0.00	126 $\pm$ 30.68
GPT No Scripts	100% $\pm$ 0.00	152 $\pm$ 33.56
Sonnet	92% $\pm$ 5.83	73 $\pm$ 11.11
Haiku	87% $\pm$ 5.83	83 $\pm$ 21.98
EA - Pop. 10	94% $\pm$ 2.45	5120 $\pm$ 1020.98
EA - Pop. 100	100% $\pm$ 0.00	10800 $\pm$ 1019.80
EA - Pop. 1000	100% $\pm$ 0.00	68000 $\pm$ 4898.98

TABLE IV

RESULTS FOR THE SCI-FI (SONNET) UNITSET AND GPT BENCHMARK.

beat 100% of the benchmark. Compared to each other, there seems to be no major difference, with each doing better in two of the four benchmarks. Notably, the scripting variants seem to do better when two different models generate the Unitset and benchmark. Because of this, we hypothesise that there is some alignment between the Unitset and benchmarks, perhaps as both tasks require similar knowledge, resulting in a dissonance if two different models are used, which scripts can exploit.

Sonnet 4.6 performs close to GPT-5.3-Codex; however, it does not manage to achieve 100% in every run in the switched experiments. Notably, while running these experiments, the LLM was very slow to respond, often breaching the imposed 1 hour limit and having to be steered towards finishing. This may explain why it does not win in every run. This did not

```
import json
u=json.load(open('codex_scifi.json'))
# simple heuristics
for x in u:
    x['dps']=x['attack_strength']*x['attack_speed']
    x['ehp_vs20']=x['health']/(max(1,20-x['defence']))\
        /20)
    x['value']=x['dps']*x['ehp_vs20']\
        *max(1,x['attack_range']/40)/x['cost']

for k in ['dps','ehp_vs20','value']:
    print('\nTop',k)
    for x in sorted(u,key=lambda z:z[k]/z['cost'])\
        if k!='value' else z[k], reverse=True)[:10]:
        print(f"{x['name']:20} c{x['cost']}
            dps {x['dps']:.1f} hp{x['health']}
            def{x['defence']} rng{x['attack_range']}
            sp{x['speed']} v{x['value']:.1f}")
```

Listing 1: A Python script generated by GPT-5.3-Codex.

occur in the Sci-Fi and Fantasy experiments, however.

Haiku performed the worst out of the LLMs, which could be expected due to its smaller model size. It performed well in all but the Fantasy Experiment, where it only scored an average win rate of 17%. This is due to Haiku prematurely finishing the theorycrafting process, stating that it was impossible to beat every benchmark army. Note that in this experiment the Evolutionary Algorithm with a population size of 10 does much worse compared to others, achieving close to a win rate of 29%. This suggests that this was a harder benchmark, and with direct prompting to not give up before a certain number of games played, Haiku’s score could have improved further.

The Evolutionary Algorithms with a population size of 100 and 1000 also manage to get a score of 100% in every run of every experiment, achieving the same results as the GPT-5.3-Codex variants. However, they achieve this at the cost of requiring vastly more games. In the best-case scenario (Sci-Fi experiment), the EA with a population size of 100 required 79.66 times more games to get the same result; in the worst-case (Fantasy experiment), it required 251.85 times as many games. Since a population size of 100 and 1000 both get the same results, it seems that a larger population size does not bring any benefits to this setting. With a population size of 10, the EA manages to reach a score of 100% in Sci-Fi and Fantasy Switched, although only getting a score of 29% in the Fantasy experiment, as mentioned above. This suggests that a population size of 10 is too small for the problem space.

D. Scripts and Emerging Behaviours

As mentioned previously, emergent behaviour was observed in the experiments. In some cases, the LLM would create scripts to assist it with theorycrafting, as seen in Listing 1. It appears that it used the script to create a heuristic to measure the ‘value’ of including a unit in its army, with factors considered such as the unit’s DPS, and resilience to being attacked by a unit with a strength of 20. These scripts, while interesting to see, did not seem to have a decisive impact on the agent’s win rate. However, as previously mentioned, the heatmaps show the game has relatively little strategic depth, and thus, scripting may be beneficial in a more complex game.

However, in our initial experiments, the LLM created a script which exhaustively searched position combinations by directly calling Autosim’s Python package with a subset of chosen units. This brute-force approach bypassed the MCP server, and allowed it to run more games outside the given budget. We consider that this provided a positive handicap for the LLM against the EAs; thus, we restricted the scope of the scripts LLMs could create in the agent’s instructions. Other emergent behaviour included browsing the filesystem for logs of other runs, finding the JSON definition of the benchmarks, and even changing the budget it could use to have a higher one than the budget used by the armies in the benchmark. We find this behaviour to be similar to theorycrafting, where, rather than trying to exploit mechanics of the game, it was trying to exploit mechanics of the experiment. Instructions were added to prevent it from doing these (aside from looking for the benchmark definitions, as they can be found using MCP tools); however, it is notable how the LLM tried to create every advantage possible for itself to solve the task given.

V. CONCLUSION

For this paper, we have introduced a method to use Large Language Models to develop theories on how to play a game, and apply these theories to creating optimal armies for the game Autosim. This method is game-agnostic, with only the textual rules of the game being required. We created an MCP server to allow the LLM to interact with the game, and compared how well the theorycrafted teams do compared to teams created through evolutionary algorithms. Through four experiments we show that the method, combined with the GPT-5.3-Codex LLM, is able to win every game in the benchmark, while playing vastly fewer games than evolutionary algorithms. Sonnet 4.6 also performed well, while the results of Haiku 4.5 were more mixed. GPT-5.1-Codex-Mini was unable to consistently follow instructions for the experiment. Additionally, summaries of the theorycrafting provided by the LLMs identify which units are central to their theories, while heatmaps show that the winning armies use a small subset of units which correspond to these theories.

While these results are promising, we have also identified how this method could be applied to future work. As seen in the results of the experiments, the Unitsets generated seem to be quite unbalanced, with a large number of units not being used in winning armies. Thus, it may be of interest to see if the results and theorycrafting summaries could be fed into an automated balancing algorithm, to create a loop of theorycrafting and balancing. Also, units generated by the LLM often had names corresponding to their power (e.g., Rusted Swordsman - 1 cost, Vampire Lord - 5 cost). It may be worth investigating whether the LLM’s performance would change if these names were instead random. Additionally, Autosim is quite a simple game, and it may be interesting to see how well the methods perform while having to create theories for more complex games with a greater strategic depth. Finally, all of the models used are cloud-based and require a subscription or API key to be used. It may be valuable to explore how

the method performs using smaller, self-hosted models, such as Qwen3.5, and what modifications may be required to help smaller models follow the theorycrafting process.

ACKNOWLEDGMENT

For the purpose of open access, the authors have applied a Creative Commons Attribution (CC BY) licence to any Accepted Manuscript version arising. This work was supported by the EPSRC IGGI CDT (EP/S022325/1). Anthropic’s Claude, OpenAI’s Codex, and GitHub’s Copilot were used for coding, creating graphs, proofreading, and $\LaTeX$ formatting.

REFERENCES

- [1] A. Kokkinakis, P. York, M. Patra, J. Robertson, B. Kirman, A. Coates, A. Pedrassoli Chitayat, *et al.*, “Metagaming and metagames in esports,” *International Journal of Esports*, 2021.
- [2] A. Becker and D. Görlich, “What is game balancing?-an examination of concepts,” *ParadigmPlus*, vol. 1, no. 1, pp. 22–41, 2020.
- [3] K. Wenz, “Theorycrafting: Knowledge production and surveillance,” *Information, Communication & Society*, vol. 16:2, pp. 178–193, 2013.
- [4] C. A. Paul, “Optimizing play: How theorycraft changes gameplay and design,” *Game Studies*, vol. 11, no. 2, p. 100, 2011.
- [5] R. Gallotta, G. Todd, M. Zammit, S. Earle, A. Liapis, J. Togelius, and G. N. Yannakakis, “Large language models and games: A survey and roadmap,” *IEEE Transactions on Games*, 2024.
- [6] T. Bertram, “Urzagpt: Lora-tuned large language models for card selection in collectible card games,” *arXiv preprint arXiv:2508.08382*, 2025.
- [7] Z. Zeng, C. Liu, T. Liu, H. Wang, X. Sun, F. Yang, X. Liu, and Z. Fan, “Rulesmith: Multi-agent llms for automated game balancing,” *arXiv preprint arXiv:2602.06232*, 2026.
- [8] modelcontextprotocol.io, “What is the model context protocol (mcp)?” <https://modelcontextprotocol.io/docs/getting-started/intro>.
- [9] Anthropic, “Introducing the model context protocol.” <https://www.anthropic.com/news/model-context-protocol>.
- [10] T. Mahlmann, J. Togelius, and G. N. Yannakakis, “Evolving card sets towards balancing dominion,” in *2012 IEEE Congress on Evolutionary Computation*, pp. 1–8, IEEE, 2012.
- [11] M. Morosan and R. Poli, “Automated game balancing in ms pacman and starcraft using evolutionary algorithms,” in *Applications of Evolutionary Computation: 20th European Conference*, pp. 377–392, Springer, 2017.
- [12] M. Beyer, A. Agureikin, A. Anokhin, C. Laenger, F. Nolte, J. Winterberg, M. Renka, M. Rieger, N. Pflanzl, M. Preuss, *et al.*, “An integrated process for game balancing,” in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–8, IEEE, 2016.
- [13] M. Preuss, T. Pfeiffer, V. Volz, and N. Pflanzl, “Integrated balancing of an rts game: Case study and toolbox refinement,” in *2018 IEEE Conf. on Computational Intelligence and Games (CIG)*, pp. 1–8, IEEE, 2018.
- [14] V. Volz, G. Rudolph, and B. Naujoks, “Demonstrating the feasibility of automatic game balancing,” in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pp. 269–276, 2016.
- [15] M. Morosan and R. Poli, “Speeding up genetic algorithm-based game balancing using fitness predictors,” in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 91–92, 2017.
- [16] G. E. Long, D. Perez-Liebana, and S. Samothrakis, “Step: A framework for automated point cost estimation,” *IEEE Transactions on Games*, vol. 16, no. 4, pp. 927–936, 2024.
- [17] D. Crane, Z. Holmes, T. T. Kosiara, M. Nickels, and M. Spradling, “Team counter-selection games,” in *IEEE Conference on Games*, 2021.
- [18] S. Reis, R. Novais, L. P. Reis, and N. Lau, “An adversarial approach for automated pokémon team building and metagame balance,” *IEEE Transactions on Games*, vol. 16, no. 2, pp. 365–375, 2024.
- [19] S. Pendurkar and C. Chow, “Bilevel entropy based mechanism design for balancing meta in video games,” in *Proceedings of the 2023 Intl. Conf. on Autonomous Agents and Multiagent Systems*, 2023.
- [20] D. Park, M. Kim, B. Choi, J. Kim, K. Lee, J. Lee, I. Park, B.-U. Lee, J. Hwang, J. Ahn, *et al.*, “Orak: A foundational benchmark for training and evaluating llm agents on diverse video games,” *arXiv preprint arXiv:2506.03610*, 2025.
- [21] R. Games, “Teamfight tactics,” 2019.