An Initial Design for Generative Game Mechanics via LLM Function Emulation

Brad Spendlove
Department of Mathematics & Computer Science
Randolph College
Virginia, USA
bspendlove@randolphcollege.edu

David Kline
Randolph College
Virginia, USA
dkline@randolphcollege.edu

Abstract

A video game's mechanics dictate the outcome of player input and world state changes. Game mechanics are functions invoked in response to game actions (often player input) that result in changes to the game state. We present an initial design for a game system that takes on the creative responsibility of determining mechanical outcomes of key game actions in a dungeon crawler video game by emulating its mechanic functions via an LLM. The system emulates those functions by prompting an LLM with a text description of the mechanic and the input to the function. The LLM's response is parsed to extract the output of the emulated function. We present an initial implementation of combat and enemy spawning with LLM function emulation replacing human-coded game mechanics. Initial results suggest that chain-of-thought prompting enables the LLM to make sensible game updates based primarily on plain-text descriptions of enemies, grounded in the game state.

Introduction

Video games are traditionally constrained by the limits of hard-coded rules, which can make complex interactions predictable or labor-intensive to design and implement. Large language models (LLMs) present an alternative paradigm: by interpreting player input and generating gameplay results in real time, LLM-driven game mechanics can enable flexible emergent gameplay experiences that may be otherwise difficult to achieve.

"A game is a voluntary interactive activity, in which... players follow rules that constrain their behavior, enacting an artificial conflict that ends in a quantifiable outcome" (Zimmerman 2004). Video games as interactive computer programs maintain a game state, consisting of all the variables relevant to the game scenario. The game state is repeatedly updated based on player input and the state itself, as depicted in Figure 1. Game mechanics are functions that map the current game state and player input to an updated game state (Sicart 2008).

This game loop paradigm is ubiquitous in game development. For example, in a 2D game, a common game state variable is the player character's (x,y) position. When the player gives directional input, the position variable is changed via some function in every iteration of the game loop. The position variable can also change based on other game state variables, such as gravity or object collisions. The game state is rendered into video and audio to communicate that state to the player, such as changing the location of the character on the screen based on its position variable.

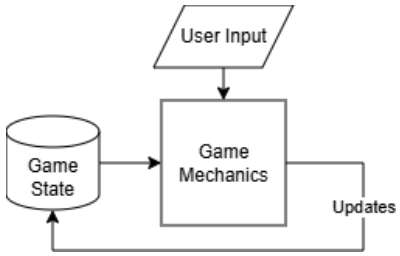

Figure 1: Fundamental game update loop.

The exact manner in which game state variables are updated based on input and other variables constitutes the game's mechanics, which the players learn and interact with. Modern game states and updates are often very complex and incorporate analog input, physics simulations, and non-player character AI. But whether a game is simple or complex, its game mechanics are what make it the game it is.

Human game developers typically hard-code game mechanic functions. We seek to replace human-written game mechanics by emulating those functions using a large language model. Specifically, we prompt the LLM with the game state and player input and ask it to generate state updates that are then parsed and applied to the game state. Prompting an LLM this way implements the state update function in a more flexible and dynamic manner than hard-coded game mechanics.

Our game system also uses the LLM to generate text descriptions of game actions, similar to games like NetHack (The NetHack DevTeam 1987). It is well known that LLMs are adept at creating text. This work focuses on how LLMs can also generate matching game state updates that are both creative and mechanically sensible.

The primary contribution of this paper is the presentation of a novel method for executing video game mechanics. This short paper presents our design and initial system implementation. Preliminary testing suggests that our game system achieves our design goals of implementing game me-

chanics by prompting an LLM with game state information instead of directly calculating game state updates through hard-coded functions. Future work will include further evaluation, including user studies and ablation testing.

Related Work

LLM-driven Games

There is much research into using LLMs to power games. Many of those systems build directly on top of LLMs' natural language capabilities to implement text-based games wherein the players' inputs to the game are natural language text. The game presented herein is not text-based. The player interacts using traditional button controls (primarily directional and selection buttons), and the game is presented visually as a character moving around a 2D grid-based representation of a dungeon. This places our system in a distinct genre from text-based games (Lee et al. 2014) and represents distinct design and gameplay approaches.

As reported in a survey by Gallotta et al. (2024), other existing research into LLM-generated game mechanics are distinct from our approach. Those systems fall into one of three categories: they use LLM prompts and responses directly as the input and output to a text-based game, they use LLMs to select and invoke hard-coded functions (Song, Zhu, and Callison-Burch 2024), or they use LLMs to create or modify source code instructions that will be executed when those methods are invoked (Nasir et al. 2025).

PAYADOR (Góngora et al. 2024) is an example of a text-based interactive fiction grounded in a persistent world state, including character description, location, and inventory. It is similar in form to AI Dungeon (Hua and Raley 2020), but achieves a more consistent game world by grounding LLM prompts and responses in the persistent game state.

LLMs with Function Calling

In tabletop role-playing games, the Game Master (GM) is responsible for maintaining and communicating a game world in the theater of the mind. Thus, the GM fulfills a role similar to the game loop in a video game. Song et al. (2024) present a system for running a tabletop role-playing game as the Game Master (GM). Their system is dialogue-based, in which players narrate their game actions and the system responds with the outcomes of those actions.

In order to maintain a consistent game state during gameplay, their system utilizes LLMs with function calling (Li et al. 2024). This enables the LLM to include invocations to externally defined functions in its responses. Those functions directly update the game state according to the parameters provided upon invocation. Gallotta et al. (2024) also employ LLMs with function calling to create game content data in dialogue with a human designer.

The system presented herein differs from this function calling approach in that there are not externally defined functions that the LLM invokes in its responses. Instead, we use the LLM to emulate the behavior of a function directly, as described above. To the best of our knowledge, this present work is the first research that applies this function emulation approach to implementing game mechanics in a typical action game with non-textual input.

Implementation

Large Language Model and Prompting

Our system uses Llama 3.3 (Grattafiori et al. 2024) as the LLM that drives the game mechanics. We use the off-the-shelf model with no fine-tuning, accessing it via the Groq API[1]. The model has 70 billion parameters and is a high-speed and cost-effective improvement over previous models. Those characteristics are desirable for a model that will run a game engine due to the potentially high volume of LLM calls for updating the game state.

Prompt engineering is a critical part of the design process for any system that uses LLMs (Giray 2023). Our game system incorporates two primary prompting patterns: chain-of-thought prompting (Wei et al. 2022), which involves breaking a reasoning task into single logical steps, and few-shot prompting (Brown et al. 2020), in which example outputs are provided in the prompt to help the LLM's response align with those examples.

Our game system makes use of chain-of-thought prompting to work through various game state updates, treating each as a single question instead of asking for the whole suite of updates at once. It uses few-shot prompting to specify an output format for the LLM's game state updates, facilitating parsing and grounding in the existing game state.

Game System

The game we implemented is a top-down 2D dungeon crawler with roguelike elements (Szabados et al. 2023), namely turn-based gameplay and grid-based movement. A turn-based game loop that only advances when the player provides input allows our system to calculate game state updates much less frequently than a real-time game loop.

Within this game design, we selected a subset of game mechanics to implement using LLM updates. All game mechanics are method invocations that modify the game state, so any of the mechanics in our game could be implemented either with traditional hard-coded methods or with LLM-emulated methods. We selected the two most impactful and defining mechanics of our game to implement this way. We call these selected mechanics the *core* mechanics, and we will refer to the other non-LLM mechanics as *simple*.

These names are not arbitrary. The set of non-LLM game mechanics we call *simple* handle common or routine interactions for which the mechanical outcome is simple to calculate. These interactions are processed into game state updates using traditional hard-coded functions. Examples include moving the player character around the grid in response to directional input and accessing the inventory menu to equip items. These workhorse interactions benefit from being lightweight and predictable.

Core mechanics are the two main mechanics that define our game: combat and enemy spawning. These have the

[1]https://console.groq.com/docs/model/llama-3.3-70b-versatile

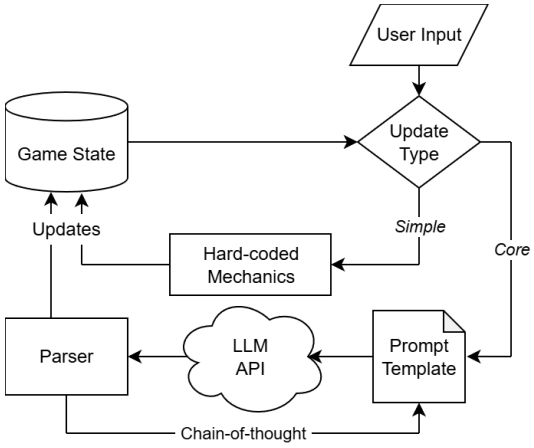

Figure 2: Our game system loop that divides mechanics into two categories: *simple* and *core*. Only the latter are processed through the LLM.

most impact on players, so implementing them with LLM-driven updates will most directly influence the gameplay experience. The exact outcome of these mechanics is unknown due to the wide variety of outputs LLMs generate. We believe this will be beneficial to the player experience, because "games require uncertainty to hold our interest" (Costikyan 2013). Combat is more interesting if it plays out differently every time, and LLM-generated enemies will have more variety than if they were drawn from a predetermined set.

Figure 2 shows our gameplay loop. When player input is registered, it is combined with the game state to determine whether the resulting game state updates are *simple* or *core*. If they are *simple*, they are processed using traditional hard-coded functions. If they are *core*, then the mechanic is processed by prompting an LLM.

Prompting

Each *core* mechanic has one or more prompt templates associated with it. Each template is filled with relevant game state data and used to prompt the LLM via an API call. The LLM's response is then parsed to extract updates that are applied to the game state.

Some interactions use a sequence of prompt templates that implement chain-of-thought reasoning. Each prompt is given to the LLM in sequence to resolve multi-step interactions, such as spawning new enemies. Game state updates are applied at the end of the chain-of-thought sequence after all relevant updates have been collected.

It is critical to the *core* game mechanic functions that both their input (current game state and player input) and output (game state updates) are grounded in the game state. Therefore, our *core* mechanic prompt templates include two features. First, the templates include variables drawn directly from the game state so that they appear explicitly in the prompt. Second, they employ few-shot prompting. The few-shot examples specify both the output format expected by our parser and the set of variables that the LLM can update for a given scenario. All of the specific prompts our system used are given in Appendix A.

The LLM's response to the templated prompt is parsed to separate the textual description of the interaction and extract the corresponding game state updates. The updates are applied to the game state variables, and the description is both used for chain-of-thought prompting and displayed to the player. A response to the combat prompt is expected to include a text description followed by two state updates: the amount of damage DEALT by the player to the enemy, and the amount of damage RECEIVED in return.

Because LLMs are trained on linguistic tokens that have no inherent numeric comparability (Qian et al. 2023), we experimented with asking the LLM for state updates using qualitative words instead of numbers. The template's few-shot examples represent combat damage with the words NONE, LOW, MEDIUM, HIGH, or FATAL instead of numbers. Our parser then interprets those words into numerical values by mapping them to a corresponding percentage of the damaged character's health. A similar approach is used to represent each combatant's current health variable in the prompt template as HEALTHY, WOUNDED, or MAIMED.

Chain-of-thought prompting is used for spawning enemy reinforcements. The LLM is prompted with a template that asks how many enemy reinforcements are summoned in the text description that results from the combat mechanic, if any. If the response is greater than zero, that number and the combat description are used to populate another zero-shot template prompting the LLM to generate names and descriptions for the resulting reinforcements. Those names and descriptions directly comprise new enemy objects.

The final prompt template in the chain gives the LLM each generated enemy description and a list of sprite names, and asks it to select an appropriate sprite, which is used to represent that enemy graphically.

Gameplay

The game mechanics described above were implemented in a top-down 2D dungeon crawler game prototype. In it, the player controls a knight in a simple dungeon, attacking any enemies they come in contact with. The game is turn-based; the game loop executes one state update iteration, called a step, each time the player moves. The player inputs are 4-direction movement, a "pass" button to skip player movement for that step, and a simple inventory screen that the player interacts with using the mouse.

The default dungeon room is initially populated with three enemies: a necromancer, a greater necromancer, and a goblin. Every step, the enemies move in a random direction unless they are adjacent to the player. If they are, they will attack the player instead. Combat is a *core* mechanic that is implemented by prompting the LLM, as described above. If the player moves into an enemy, combat will involve the player and enemy attacking each other.

The game state changes resulting from each combat are generated and applied as described above. The text description is displayed to the player, along with the amount of damage each combatant dealt and their remaining health. If new enemies are spawned via that chain-of-thought-prompted mechanic, they are added to the enemy list in the

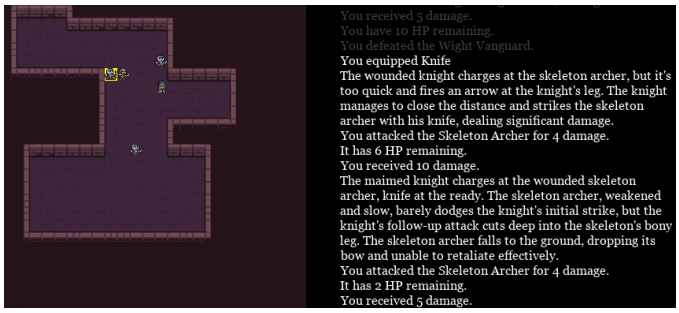

Figure 3: Cropped screenshot showing combat between the player character (highlighted in yellow) and an LLM-generated enemy. (Portion of text dimmed for emphasis.)

game state. They appear in the dungeon room and will move and interact with the player.

The player can open a simple inventory screen to equip different pre-loaded weapons. Those items are also stored with only a name and description. Any differing effects of the weapons are decided by the LLM when generating state updates from a combat encounter. In future work, we will explore how items can be generated via the LLM, similar to how enemies are spawned in the current system.

Discussion

Initial testing suggests that our game system with LLM-emulated mechanic functions exhibits novel and interesting gameplay. More testing and evaluation are needed, but we present some preliminary findings here.

Natural Language Object Representation

Our system can generate mechanical outcomes based solely on simple text descriptions of the player and enemies. When enemies are generated via the post-combat chain-of-thought process, the resulting objects store only a name, a text description, and a selected sprite image. Notably, those objects contain no explicit gameplay logic.

The name and description can then be fed back into the LLM combat mechanic when the player encounters those enemies on later steps. The results of combat with those enemies are shaped entirely by their text descriptions, replacing customized hard-coded logic that a human game developer would otherwise have to implement for each enemy. The LLM is free to draw on its training data to generate a wide variety of enemies, then implement implicit combat mechanics appropriate to each.

Figure 3 shows a screenshot of our prototype game, including the player character highlighted in yellow and several enemies. The UI includes text descriptions of two rounds of combat with the skeleton enemy next to the player (i.e., the player gave two inputs to move into the enemy, each resulting in a combat encounter).

That skeleton enemy was previously generated by the LLM during combat with a now-defeated necromancer enemy. The LLM named the generated enemy "Skeleton Archer" and gave it the following description: "A skeleton armed with a bow and quiver of arrows, providing ranged support to the necromancer." Finally, the LLM chose an appropriate sprite from a pre-loaded list, and the new skeleton enemy was added to the game state.

Later, when the player fought that enemy, the combat description the LLM generated clearly reflects characteristics of a skeleton archer (as seen in the text output), including the skeleton attacking with a fired arrow and the player's attack "cut[ting] deep into the skeleton's bony leg".

Combat Outcomes

We conducted experiments to test how the game resolves various combat scenarios. To isolate the tests from other aspects of the game, these were invocations of the emulated combat mechanic function using constructed game state data. The full results are reported in Appendix B.

The results of these tests demonstrate the LLM-emulated combat mechanic's ability to generate appropriate outcomes for the different scenarios, grounded in the game state variables. Combat against a weak enemy commonly resulted in dealing high damage and receiving little in return. When fighting a strong enemy, the player consistently dealt less damage and took the highest damage out of all scenarios. A maimed player dealt less damage and took more than a wounded or healthy player. A pair of tests on the effects of different weapon descriptions, e.g. a small knife compared to a large dragon-slaying sword, did not reflect the expected difference in state updates.

A notable emergent behavior is that across all scenarios, the player never dealt zero damage (NONE) and never received more than MEDIUM damage. This represents an implicit game design decision implemented by the LLM combat mechanic, apparently to bias the game toward player success. This is most apparent in a scenario involving a strong enemy; the LLM reduced the player's damage output and increased the enemy's, but neither exceeded those apparent bounds. The LLM was given no instructions in this regard and chose to generate updates according to those rules of its own volition.

This suggests that our game system is taking on creative responsibilities regarding the game's mechanics. Future work will more thoroughly evaluate the system and these claims.

Creative Responsibility

The creative artifact of our game system is a function that maps a game state and player input to an updated game state. Ritchie (2007) proposes a set of three criteria for assessing creative artifacts: typicality, novelty, and quality.

Our system's approach to implementing that update function results in novel gameplay interactions. Because the mechanics are not hard-coded, they can take dynamic forms that change during gameplay. The value of our system's resulting gameplay remains to be evaluated more rigorously, but at present it appears to achieve a reasonable baseline of functional and interesting gameplay. Grounding the LLM in game state data directly enhances the typicality of our system's creative artifacts and helps it avoid creating low-value artifacts such as nonsensical game state updates.

References

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33:1877–1901.

Costikyan, G. 2013. *Uncertainty in games*. Mit Press.

Gallotta, R.; Todd, G.; Zammit, M.; Earle, S.; Liapis, A.; Togelius, J.; and Yannakakis, G. N. 2024. Large language models and games: A survey and roadmap. *IEEE Transactions on Games*.

Gallotta, R.; Liapis, A.; and Yannakakis, G. 2024. Consistent game content creation via function calling for large language models. In *2024 IEEE Conference on Games (CoG)*, 1–4. IEEE.

Giray, L. 2023. Prompt engineering with chatgpt: a guide for academic writers. *Annals of biomedical engineering* 51(12):2629–2633.

Góngora, S.; Chiruzzo, L.; Méndez, G.; and Gervás, P. 2024. Payador: A minimalist approach to grounding language models on structured data for interactive storytelling and role-playing games. In *Proceedings of the 15th International Conference on Computational Creativity*. Association for Computational Creativity.

Grattafiori, A.; Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Vaughan, A.; et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Hua, M., and Raley, R. 2020. Playing with unicorns: Ai dungeon and citizen nlp. *DHQ: Digital Humanities Quarterly* 14(4).

Lee, J. H.; Karlova, N.; Clarke, R. I.; Thornton, K.; and Perti, A. 2014. Facet analysis of video game genres. *IConference 2014 Proceedings*.

Li, Z.; Chen, Z.; Ross, M.; Huber, P.; Moon, S.; Lin, Z.; Dong, X. L.; Sagar, A.; Yan, X.; and Crook, P. A. 2024. Large language models as zero-shot dialogue state tracker through function calling. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 8688–8704.

Nasir, M. U.; Li, Y.; James, S.; and Togelius, J. 2025. Mortar: Evolving mechanics for automatic game design. *arXiv preprint arXiv:2601.00105*.

Qian, J.; Wang, H.; Li, Z.; Li, S.; and Yan, X. 2023. Limitations of language models in arithmetic and symbolic induction. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 9285–9298.

Ritchie, G. 2007. Some empirical criteria for attributing creativity to a computer program. *Minds and Machines* 17(1):67–99.

Sicart, M. 2008. Defining game mechanics. *Game studies* 8(2):1–14.

Song, J.; Zhu, A.; and Callison-Burch, C. 2024. You have thirteen hours in which to solve the labyrinth: Enhancing ai game masters with function calling.

Szabados, G. N.; Bácsné Bába, É.; Fenyves, V.; Bács, Z.; Molnár, A.; Ráthonyi, G.; Rizwan, H.; and Orbán, S. G. 2023. Roguelike games: The way we play. *International Journal of Engineering and Management Sciences* 7(4):80–92.

The NetHack DevTeam. 1987. Nethack. `https://www.nethack.org`. Computer software.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Xia, F.; Chi, E.; Le, Q. V.; Zhou, D.; et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35:24824–24837.

Zimmerman, E. 2004. Narrative, interactivity, play, and games: Four naughty concepts in need of discipline. *First person: New media as story, performance, and game* 154.

Appendix A: Prompts

All prompts are templated to be populated with game state data, indicated by variable names surrounded by curly braces.

Combat Prompt (second few-shot example omitted for brevity):

```
Resolve this video game combat scenario,
  given the name and description of the
  player character and enemy to engage.
  Adjust the strength of the player and
  enemy based on their status: HEALTHY,
  WOUNDED, or MAIMED.
Keep the output brief, within 4 sentences if
  possible.
Return a damage value of LOW, MEDIUM, or
  HIGH. If the player or enemy is defeated,
  the dealt or received value should be
  FATAL.
Example 1:
Player class: Knight
Status: HEALTHY
Class description: A knight, armed with a
  spear. A powerful melee combatant.
Enemy: Skeleton Grunt
Status: HEALTHY
Enemy description: A weak skeleton, armed
  with a knife.
Output: The skeleton grunt viciously
  slashes at the knight, but its knife
  cannot penetrate his armor. The knight
  retaliates by bashing the grunt with the
  pole of his spear, knocking it to the
  ground. DEALT: MEDIUM, RECEIVED: NONE
Example 2:
...
Using the format from the examples, run
  a combat scenario with a {enemy.status}
  enemy {enemy.name}, with the description
  {enemy.description}, attacking a
  {player.status} player {player.name}, with
  the description {player.description}.
```

Enemy Generation Chain-of-thought Prompt 1: Identifying Reinforcements:

```
Read this scenario {scenario}. If it
  explicitly states the enemy has called
  reinforcements, determine the number of
  enemies to be summoned.
If enemies are summoned, the number of
  enemies should be 1 at minimum, or 3
  at maximum. If no reinforcements are
  directly mentioned, provide 0 for the
  number of each enemy.
Format the enemy count like so:
Enemies: 2
```

Enemy Generation Chain-of-thought Prompt 2: Generating Enemy Names and Descriptions:

```
Based on the provided combat scenario
  {scenario} where the enemy calls for
  reinforcements, create appropriate enemies
  to be summoned.
Here are some example enemies and their
  descriptions:
Format the enemies like so:
Enemy 1: Skeleton Grunt, A weak skeleton,
  armed with a knife.
Enemy 2: Reaper, The skeleton of a strong
  warrior, armed with a scythe.
Create a maximum of {count} enemies.
```

Enemy Generation Chain-of-thought Prompt 3: Selecting Enemy Sprite:

```
Given the name {name} and description {desc}
  of an enemy, select the sprite from
  {sprites} that suits them the best.
```

Appendix B: Combat Test Results

Scenario	Weak Enemy		Strong Enemy		Wounded	
Damage	Dealt	Received	Dealt	Received	Dealt	Received
NONE	0	4	0	0	0	0
LOW	0	6	10	0	1	8
MEDIUM	1	0	0	10	8	2
HIGH	1	0	0	0	1	0
FATAL	8	0	0	0	0	0

Scenario	Maimed		w/ Knife		w/ Slayer	
Damage	Dealt	Received	Dealt	Received	Dealt	Received
NONE	0	0	0	1	0	0
LOW	7	5	0	9	0	10
MEDIUM	3	5	6	0	7	0
HIGH	0	0	2	0	3	0
FATAL	0	0	2	0	0	0

The outcomes of testing the LLM-emulated combat mechanic function in several scenarios, each consisting of different game state variables. Each column records the results of running the test 10 times, recording how many times each damage description was returned for each scenario. Recall that those qualitative words were used in place of numerical results to better align with LLM representations. Damage dealt and received are from the player's perspective (i.e., *"dealt" by the player* and *"received" by the player*).

Each scenario uses the same player description: "A knight armed with [weapon]". The player has full health in all scenarios except for the Wounded and Maimed scenarios, in which the player has 50% health and 5% health, respectively. The default weapon description is "a well-forged longsword". The w/ Knife scenario replaces this with "a well-maintained knife, very sharp, but it has short range", and the w/ Slayer scenario uses "a massive sword, empowered by slain dragons". The enemy in the Weak scenario is described as "a weak, pathetic goblin", the enemy in the Strong scenario is "a huge, indestructible granite construct", and the rest use the default of "a small but fierce goblin warrior".