Language Model-Based Agents to Learn Policy for Text Based Markov Decision Processes

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have demonstrated strong capabilities in solving various reasoning tasks but typically struggle when directly asked to solve complex Markov Decision Process (MDP) problems due to their sequential and algorithmic nature. In this work, we propose **OptiAct**, an algorithmic reasoning framework that leverages LLMs by decomposing MDP problems into structured subtasks such as component extraction, mathematical formulation, code generation, and execution to select the **Opti**mal **Act**ions. Our approach is a cooperative multi-agent system that decomposes the task into verification, component extraction, formulation, and code generation. This modular approach enables validation at each stage while maintaining interpretability throughout the solution process. To systematically evaluate this method, we create real-world decision-making problems with varying complexity from different sources and evaluate our pipeline on five different large language models. Experiments demonstrate that algorithmic decomposition significantly enhances LLMs' effectiveness in solving finite-horizon MDPs, highlighting the necessity and benefits of structured reasoning over direct unstructured solution generation.

1 Introduction

Markov Decision Process (MDP) is a fundamental framework for modeling sequential decision-making under uncertainty and essential to artificial intelligence, operations research, and reinforcement learning (Puterman, 1994). MDPs provide a mathematical structure to formalize problems involving states, actions, rewards, and transitions, enabling the design of optimal decision that maximizes cumulative rewards over time. Formulating and solving MDPs often requires significant domain expertise and algorithmic knowledge, making them less accessible to non-experts (Yu et al., 2021; Shakoor et al., 2016). Consequently, there is a growing need for autonomous agents that can reason and act in dynamic environments without relying on human expertise.

Large language models, trained on vast corpora of text and structured data, have demonstrated emergent capabilities to solve a wide range of reasoning tasks (Wei et al., 2022). These models have shown proficiency in solving equations, performing algebraic manipulations, and addressing advanced topics such as calculus and linear algebra (Ferreira and Freitas, 2020; Yu et al., 2023). Recent studies have demonstrated that LLMs can effectively parse optimization problems, identify constraints, and propose solutions in a structured manner (Ramamonjison et al., 2023; AhmadiTeshnizi et al., 2023; Yang et al., 2025). This type of problems are static decision-making problem where the goal is to find the best solution from a set of feasible solutions, often subject to constraints. Contrary to this, sequential decision-making problems involve dynamic environments where the agent must make a series of decisions over time, considering the impact of current actions on future states and rewards. Recent advancements have highlighted the potential of LLMs to perform action-oriented reasoning across dynamic environments. Researchers have demonstrated that LLMs can parse environmental signals, generate context-aware plans, and execute decisions using natural language as a unified interface (Yao et al., 2020; Schick et al., 2023; Yao et al., 2022; Brohan et al., 2023; Lin et al., 2023). These systems are typically deployed in benchmark environments (Shi et al., 2017; Shridhar et al., 2021; Li et al., 2022; Liu et al., 2018), where task-solving involves multi-step implicit reasoning, grounded state tracking, and tool use. Most of these approaches require few shot examples or post training for the LLM to learn from the given expert trajectories. However, their potential for solving tabular MDPs in a structured and algorith-

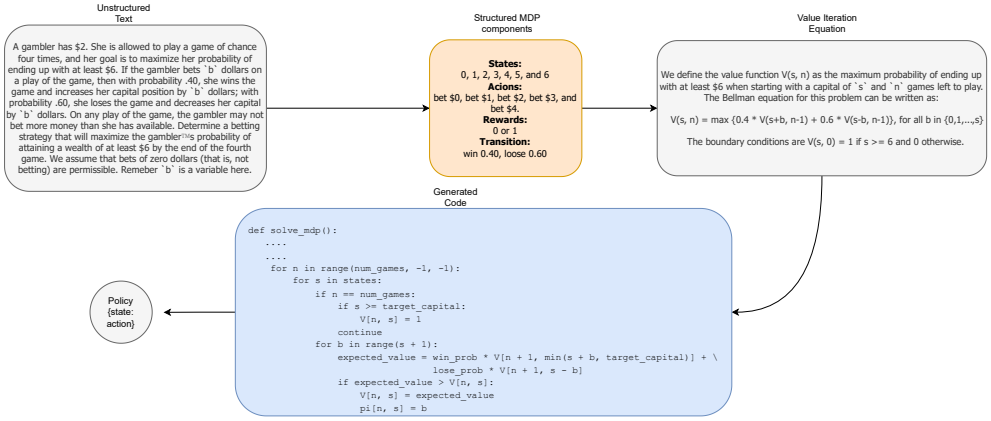


Figure 1: Illustration of the stages involved in formulating and solving a Markov Decision Process (MDP) problem from an unstructured text description. From text input, the essential MDP components—states, actions, rewards, and transitions are identified. These components are then translated into a formal, structured representation suitable for algorithmic processing such as value iteration, and expressing it through the Bellman optimality equations. Finally, this mathematical formulation is converted into executable code, allowing the derivation of an optimal policy.

mic manner remains unexplored.

As indicated in Figure 1, to formulate a decision making problem from an unstructured description, as humans we first need to describe the basic components of a Markov decision process - states, actions, rewards, transitions, discount factor, and time horizon. Then choose value iteration or policy iteration to design the algorithm (Sutton and Barto, 2018), and finally write code to get the optimal policy. This end-to-end transformation from unstructured narrative to structured solution highlights the complexity of the reasoning required and motivates the use of language models to automate these steps. Therefore, we ask - *Are LLMs capable of understanding, formulating, and generating dynamic decision-making policies within a structured MDP framework without any post training?*

To address these challenges, we propose **OptiAct**, a structured algorithmic reasoning framework tailored for decomposing and solving text based MDPs with LLMs. OptiAct breaks down the problem into key stages:

1. *Component Extraction:* Identify states, actions, transitions, rewards, and horizon from natural language descriptions.

2. *Formalization:* Translate the extracted components into a precise mathematical specification of the MDP, and map them onto a corresponding value iteration algorithm.

3. *Execution:* Compile the structured MDP specification into executable Python code, run it using a standard interpreter, and store the output for evaluation. If execution fails, a feedback loop is triggered to automatically revise and rerun the code until correctness is achieved or a maximum number of attempts is reached.

We evaluate OptiAct on a dataset of diverse real-world MDP problems sourced from classic texts (Winston, 2004; Sutton and Barto, 2018), each annotated with ground-truth specifications. In addition, we evaluate OptiAct on text-based control tasks extracted from MiniWob and TextWorld (Yao et al., 2017; Côté et al., 2018). Our experimental results show that structured decomposition of MDP problems significantly improves LLM performance compared to a direct chain of thought prompting.

2 Related Works

Sequential decision-making with language models has primarily focused on high-dimensional control tasks and robotics applications. SAYCAN (Brohan et al., 2023) and Inner Monologue (Huang et al., 2022) generate action plans in embodied environments, while ATARI-GPT (Waytowich et al., 2024) evaluates LLM performance on reinforcement learning benchmarks. These approaches typically rely on learning-based methods or require extensive fine-tuning with expert demonstrations. (Brooks et al., 2023) demonstrate that large language models can implement policy iteration by simulating the Bellman backup process through few-shot prompting alone. Liang et al. (Liang et al., 2023) generate executable policy code from natural language specifications, and Zhu et al. (Zhu et al.,

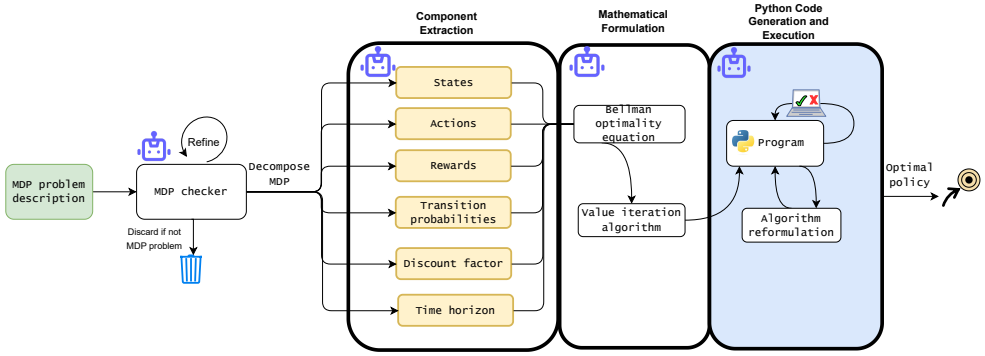


Figure 2: Overview of the proposed OptiAct framework for solving MDP problems using a sequence of four specialized LLM agents. (1) **MDP Checker** verifies whether the problem conforms to the formal structure of an MDP and removes ambiguity; (2) **Component Extraction** parses the unstructured text into structured representations of MDP components such as states, actions, rewards, and transitions using object-oriented data structures; (3) **Mathematical Formulation** generates the Bellman equations and value iteration algorithm based on the structured components; (4) **Code Generation and Execution** translates the mathematical formulation into executable Python code and runs it to derive the optimal policy.

2024) investigate policy refinement through LLM-guided reinforcement learning. Du et al. (Ajay et al., 2023) focus on using LLMs to define intermediate subgoals for hierarchical planning. However, these methods focus on direct policy generation rather than the structured component extraction and mathematical formulation that characterizes MDP problems.

Concurrently, in the domain of supply chain and operations research, recent work has explored using LLMs to bridge the gap between natural language queries and optimization.OPTIGUIDE (Li et al., 2023) a framework that uses LLM to bridge the gap between supply chain automation and human comprehension that takes plain texts as queries. (Quan and Liu, 2024) introduces a multi-agent inventory management system by leveraging the zero-shot capabilities of large language models. Several similar recent works (AlMahri et al., 2024; Aghaei et al., 2025) explored how LLMs can enhance the decision making process. OPTIMUS, proposed by (AhmadiTeshnizi et al., 2023), is a framework for solving linear and mixed-integer optimization problems from natural language using a standardized prompt format called SNOP (Structured Natural Language Optimization Problem). While SNOP effectively organizes problem descriptions, solver intent, and output structure, it is still a free-form text template which requires regular expression to match and extract a specific field. Unlike schema-driven tools, this lacks structured reasoning and type enforcement which provides better guidance to the LLM's internal decision-making process (Chen et al., 2024; Tang et al., 2023). Moreover, these approaches focus on static decision making and are not suitable for text based dynamic decision making. To address this gap, we propose a more principled prompting strategy that embeds structure and type enforcement into the LLM's reasoning process to solve dynamic decision making problems.

3 Background

3.1 Markov Decision Process

For sequential decision making, we consider finite horizon Markov decision problem (Bertsekas, 2012; Bellman, 1957), formally defined as as tuple, $\mathcal{G} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma, \mathcal{H})$ where $\mathcal{S}$ and $\mathcal{A}$ denote the finite set of states and actions and $\gamma \in [0, 1]$ is the discount factor. Considering a single agent starting at initial state s, at every decision epoch, the agent observes its current state's status i.e local observations $o_i \in \Omega$ generated by the function $O(s, a) : \mathcal{S} \times \mathcal{A} \to \Omega$ and chooses an action $a \in \mathcal{A}$ from the finite action set. Once a new action is taken the system transits to a new state s' according to the transition probability matrix or dynamics function $\mathcal{T}(s'|s, a) : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. The agent receives a reward for each action it takes in each state defined as $r(s, a)$ and its goal is to learn a policy π that maximizes the average reward $\mathbb{E}^{\pi}[\sum_{t=0}^{N} \gamma^t r(s_t, a_t)]$ where N is the final time step and terminal state is the boundary condition with reward $r_N(s_N)$. Given a state, the learned policy $\pi(a|s)$ represents the distribution of

actions. In an offline setting, the policy is learned from a static dataset that consists of sequences of actions observations and actions (i.e trajectories) $[o_1, a_1, o_2, a_2, \ldots, o_N, a_N]$. For fixed horizon length Markov decision process becomes finite known as tabular MDP and we can find the optimal policy deterministically (Sutton and Barto, 2018). Following (Pashenkova et al., 1996), we utilize value iteration algorithm and prompt LLMs to solve text based tabular MDP problems using dynamic programming.

3.2 Large language models

General purpose language models, commonly known as foundation models, have capabilities that can be adapted to solve complex downstream tasks without any gradient update. Given the emergent capabilities of this models to solve novel tasks (Wei et al., 2022), it can be used to solve optimization problems, scheduling problems, and mathematical reasoning problems (AlMahri et al., 2024; AhmadiTeshnizi et al., 2023; Li et al., 2023). Since these models are not trained to solve specific problems, we need to condition the model on a few examples or task descriptions to provide instructions on how to solve the task (Brown et al., 2020). In contrast, LLMs can leverage *zero-shot chain-of-thought* (CoT) reasoning (Kojima et al., 2022) to decompose tasks into intermediate steps without explicit training examples. For complex mathematical reasoning tasks, *chain-of-expert* (CoE) LLMs (Wang et al., 2025) are used to decompose the tasks. Building on these ideas, we propose a framework that applies structured task decomposition to the domain of algorithmic decision-making.

4 Method

Pipeline Overview and Design Rationale. Solving MDPs from unstructured text requires identifying key components, constructing the Bellman equations, and implementing an appropriate algorithm. Rather than prompting a single LLM to perform all steps, we design a modular pipeline with four dedicated agents: one each for MDP verification, component extraction, mathematical formulation, and code generation. An overview of the pipeline is shown in Figure 2. The modular design addresses several key challenges in LLM-based problem-solving. *First*, it enables systematic validation at each stage, reducing error propagation through the pipeline. *Second*, it provides interpretable intermediate outputs that can be analyzed and debugged. *Third*, it allows for targeted improvements in specific components without affecting the entire system. Finally, it can be easily extended for different foundation models for each individual component.

Text to MDP Conversion Process. The first stage of the pipeline converts a problem description into a structured MDP instance through two sequential steps by two different agents.

1. **MDP Verification and Validation.**

   The MDP Checker agent with a self-play mechanism verify whether a natural language problem description contains the essential components of a Markov Decision Process: states, actions, transitions, rewards, time horizon. The agent performs a completeness check by identifying and evaluating each MDP component in the text. When ambiguity is detected in the problem description, the self-play mechanism is activated to refine the problem (Madaan et al., 2023). In this process, the LLM acts as both a validator and a critic: it first identifies ambiguous elements and then generates a refined version of the problem description that resolves these ambiguities.

2. **Structured Component Extraction.** For a valid MDP problem, the component extraction agent transforms unstructured problem descriptions into a formal MDP tuple $G = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma, \mathcal{H})$, with each element represented in a structured, object-oriented schema. This extraction process includes identifying the state space, defining valid actions per state, mapping transition probabilities $\mathcal{T}(s' \mid s, a)$, specifying reward functions $\mathcal{R}(s, a, s')$, and estimating parameters such as the discount factor γ and horizon H. Each extracted component is encoded using custom data structures (e.g., `Transition` object). These components are assembled into a unified schema, `MDPComponents` as shown in List 1 that enables modular and verifiable representations. A complete definition of this is given in the appendix.

Listing 1: Schema definition for `Transition` and `Reward` objects in the `MDPComponents` data structure.

```
title: MDPComponents
$defs:
```

```
Transition:
  type: object
  properties:
    current_state: {type: string}
    action: {type: string}
    next_state: {type: string}
    probability:
      anyOf:
        - type: number
        - type: integer
  required:
    - current_state
    - action
    - next_state
    - probability
```

Mathematical Formulation and Algorithm Generation. Once the MDP components are extracted, we convert them into a corresponding algorithmic formulation. To achieve this, the Mathematical Formulation agent takes the structured input and generates the value iteration algorithm. Leveraging the structured MDP schema, the agent completes a template of the value iteration algorithm given in the prompt by filling in problem-specific variables, setting the initial and termination conditions, and ensuring the resulting formulation is both mathematically sound and consistent with the input data. This results in a fully parameterized pseudo-algorithm, which the code generation agent can directly translate into runnable code as shown in Algorithm 1. This structured formulation acts as a canonical representation of the value iteration logic, where the abstract placeholders are instantiated with problem-specific variables extracted from the description of the problem. By enforcing a consistent schema over the MDP components and their relationships, our method eliminates ambiguity and ensures that the resulting pseudo-algorithm supports compositional reasoning and downstream code generation.

Algorithm 1: Pseudocode generated by formulation agent for gambler's problem

Input: $S = \{0 \dots 6\}$; $A = \{0 \dots 4\}$; γ=1.0; ϵ=0.01
Output: $v(s)$: value function; $\pi(s)$: optimal policy
foreach $s \in S$ **do**
  $v(s) \leftarrow \mathbb{1}[s = 6]$
repeat
  foreach $s \in \{0 \dots 5\}$ **do**
    $v_{\text{new}}(s) \leftarrow \max_{b \le s} \left(0.4 \cdot v(s{+}b) + 0.6 \cdot v(s{-}b)\right)$
  $v \leftarrow v_{\text{new}}$
until $\max_s |v_{new}(s) - v(s)| < \epsilon$;
foreach $s \in \{0 \dots 5\}$ **do**
  $\pi(s) \leftarrow \arg\max_{b \le s} \left(0.4 \cdot v(s{+}b) + 0.6 \cdot v(s{-}b)\right)$

Code Generation and Execution Framework. Given the structured MDP specification and the corresponding pseudocode formulation, the code generation agent synthesizes executable Python code that solves the MDP via value iteration. The agent performs step-wise translation of mathematical constructs into programmatic logic, ensuring semantic alignment with the Bellman update equations. The generation process includes: (1) mapping abstract MDP components (states, actions, transitions, rewards, discount factor) into Python variables and data structures; (2) implementing dynamic programming logic with convergence criteria or iteration bounds (defaulting to $\gamma = 0.99$ and 10,000 iterations if unspecified); and (3) formatting the output into a well-structured output file that captures the optimal policy, π and value function, v. The generated code is wrapped in markdown formatting for consistency and rendered for immediate execution. This prompt-driven pipeline emulates the workflow of human solvers: deriving the algorithm from first principles, coding it correctly, and producing verifiable outputs. If execution fails, a feedback loop is triggered to automatically revise and rerun the code until correctness is achieved or a maximum number of attempts is reached.

Algorithm 2: OPTIACT: End–to–End Pipeline for Text Based MDP Problems

Input: Natural–language description $\mathcal{D}$
Output: Python source code $\mathcal{C}$ that prints the optimal policy
function OptiAct($\mathcal{D}$, $maxRetry$=8):
  /* 1. Verify the description is a finite horizon MDP */
  if ***not*** MDPCHECKER($\mathcal{D}$) **then**
    return Non-MDP problem
  /* 2. Extract canonical MDP components */
  $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma, H) \leftarrow$ EXTRACTCOMPONENTS($\mathcal{D}$)
/* 3. Build Bellman formulation */
$\mathcal{F} \leftarrow$ FORMULATEBELLMAN($\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma, H$)
/* 4. Render formulation as executable Python */
$\mathcal{C} \leftarrow$ RENDERPYTHON($\mathcal{F}$)
/* 5. Execute code with self-repair loop */
for $i \leftarrow 1$ **to** $maxRetry$ **do**
  EXEC($\mathcal{C}$) ; // run in sandbox
  return $\mathcal{C}$; // success
  (Exception e) $\mathcal{C} \leftarrow$ AUTOFIX($\mathcal{C}, e$)
return "Max retries exceeded"

Solution Verification and Quality Assurance. Given the generative nature of large language models, verifying the accuracy of a complex reasoning task is challenging. Our pipeline addresses this through a comprehensive verification system that operates at multiple levels.

Component-Level Validation. Using the structured return format of MDP components as shown in Listing 1, we invoke an evaluation pipeline to validate the correctness of states, actions, rewards, and transitions against the annotated ground truth solution.

Execution-Level Validation. For the code execution phase, we implement several validation mechanisms: syntax checking to ensure generated

code is syntactically correct; runtime validation to verify that code executes without errors; performance monitoring to ensure execution completes within reasonable time limits and output verification to check that results are complete to compute the accuracy metrics.

Feedback and Iteration. If the code execution fails or the accuracy of MDP components extraction does not meet the threshold, a feedback loop is added to catch the error and fix it by prompting the large language model again. This process continues until the code execution is successful or a certain number of attempts is passed.

5 Experiments

We consider different complexity levels of MDP problems to evaluate OptiAct. To quantify the difficulty of solving a Markov Decision Process (MDP), we define a complexity score based on the size of its state and action spaces. Given a state space of size $|S|$ and an action space of size $|A|$, the score captures the decision space growth using a logarithmic scale:

$$\mathcal{C}(S, A) = \log_{10}(|S| \cdot |A| + 1) \quad (1)$$

We use this score to categorize problems into *Low*, *Medium*, and *High* complexity tiers, enabling consistent benchmarking of LLMs on structured decision-making tasks. We provide the dataset categories and all prompts used in the experiment in the Appendix section.

Dataset. We organize our dataset into two categories:

1. **Real-World Text-Based MDPs.** A collection of 21 decision-making problems from standard textbooks (Winston, 2004; Sutton and Barto, 2018), covering domains such as inventory control, hospital management, and gambling (Table 1 in Appendix). Problems are expressed in natural language and annotated with ground-truth MDPs. We augment the dataset using LLM-generated variants, resulting in 3 *Low*, 13 *Medium*, and 5 *High* complexity problems.

2. **Text-Based Games.** This set includes 23 MiniWoB tasks and 12 TextWorld tasks, reformulated into natural language decision problems. Tasks vary in complexity (0.4771 to 1.6128), with 12 classified as *Low* and 23 as *Medium* (Table 2 in Appendix).

Evaluation Models and Metrics. We conduct experiments of five different LLMs, namely `o1` (Wang et al., 2024), `o3-mini` (OpenAI, 2024), `gpt-4o` (OpenAI et al., 2024), `llama3.3-70B` (Dubey et al., 2024) and `deepseek-r1` (DeepSeek-AI et al., 2025). To measure the performance, we measure the accuracy of each MDP component extraction and structured output accuracy. Structured output accuracy indicates whether the model can provide the MDP components in our desired format in the initial step of our pipeline. To account for the generative variability of LLMs, we allow each model a maximum of 8 attempts per problem in both the formulation and code execution phases. Similarly, the code generation step is permitted up to 8 retries to produce valid, executable Python code. If a model fails to meet the threshold within these attempts, the trial is marked as unsuccessful. We use all five models as judge and report the variances in their evaluation process.

Results on Text-Based MDP Problems. Table 1 compares models on solving real-world text-based MDPs by extracting structured components—state, action, transition, and reward. `gpt-4o` solved 19 problems with 0.90 structured output accuracy. `o1`, `o3-mini`, and `deepseek-r1` solved all 21 problems with perfect output structure (1.00). Among them, `o3-mini` achieved the highest accuracy in state (0.89), transition (0.71), and reward (0.76), making it the best overall performer. `deepseek-r1` led in action extraction (0.89), and `o1` performed well overall but showed slightly lower transition (0.69) and reward (0.65) accuracy. `llama3.3-70B` showed the weakest performance, solving only 18 problems with a structured accuracy of 0.86.

As shown in Table 2, all models had up to 8 retries, with a 0.70 accuracy threshold. `o3-mini` had the highest formulation success (76.19%) with 2.90 attempts on average. `llama3.3-70B` achieved the highest code execution success (80.95%) with just 1.19 attempts, demonstrating reliable code generation. `deepseek-r1` offered balanced performance, while `gpt-4o` and `o1` showed moderate and mixed results, respectively. Given `llama3.3-70B`'s strong performance in code execution and `o3-mini`'s strength in formulation, our modular pipeline can be further optimized by assigning different LLMs to specialized roles across stages. In addition, we compare the pass@k performance of different models to assess their ability to formulate MDP and generate correct code within multiple attempts. As shown in Figure 3(a) model

Model	Solved Problems	Struct. Out. Acc.	Average Accuracy				
			State	Action	Transition	Reward	Policy
gpt-4o	19	0.90	0.82 ± 0.03	0.84 ± 0.04	0.69 ± 0.05	$\underline{0.70} \pm 0.01$	0.65 ± 0.03
o1	21	1.00	$\underline{0.88} \pm 0.02$	0.87 ± 0.02	0.69 ± 0.02	0.65 ± 0.05	0.56 ± 0.04
o3-mini	21	1.00	$\mathbf{0.89} \pm 0.02$	$\underline{0.88} \pm 0.01$	$\underline{0.71} \pm 0.02$	$\mathbf{0.76} \pm 0.03$	$\underline{0.69} \pm 0.02$
deepseek-r1	21	1.00	$\underline{0.88} \pm 0.01$	$\mathbf{0.89} \pm 0.02$	$\mathbf{0.75} \pm 0.03$	$\underline{0.70} \pm 0.01$	$\mathbf{0.74} \pm 0.02$
llama3.3-70B	18	0.86	0.77 ± 0.03	0.78 ± 0.05	0.55 ± 0.04	0.57 ± 0.02	0.52 ± 0.01

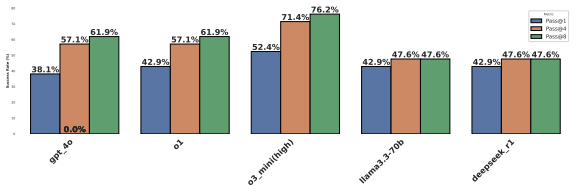
Table 1: Comparison of LLM performance across different scoring metrics. **Bold** numbers represent the best model, and underlined numbers indicate the second best.

Model	Formulation		Code Execution	
	Success Rate	Avg Attempts	Success Rate	Avg Attempts
gpt-4o	61.9%	3.05	<u>57.14%</u>	<u>3.24</u>
o1	61.9%	3.71	23.81%	6.33
o3-mini	**76.19%**	**2.90**	42.86%	5.0
deepseek-r1	<u>66.67%</u>	<u>2.38</u>	52.38%	4.57
llama3.3-70B	47.62%	1.10	**80.95%**	**1.19**

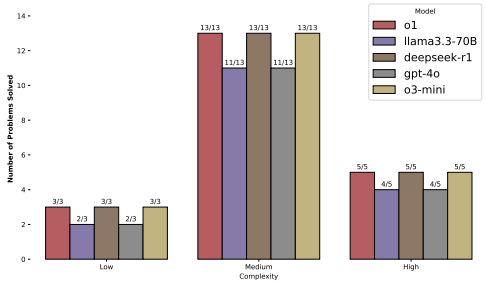
Table 2: Comparison of LLM formulation success rates and code execution success rates.

performance keeps increasing as we provide more attempts. We conduct a stratified evaluation of language model performance across low, medium, and high complexity MDP problems to assess robustness under varying task difficulty. Figure 3(b) presents the number of correctly solved problems per model in each complexity tier. Models such as `o1`, `deepseek-r1`, and `o3-mini` exhibit strong and consistent performance across all tiers, while `llama3.3-70B` and `gpt-4o` show noticeable drops in medium and high complexity settings. This suggests that reasoning-capable models are better equipped to solve MDP problems effectively.

Results on Text-Based Games. For this dataset, as illustrated in Figure 2, we explore a multi-agent prompting strategy and compare it against two baselines: single-agent prompting and unstructured prompting. This comparison highlights the benefits of modular agent design in complex decision-making tasks. In table 3, we compare model performance across three prompting settings—OptiAct (multi-agent and single-agent) and unstructured prompting—on 35 text-based decision tasks. The multi-agent OptiAct setting with `o3-mini` achieves the highest accuracy across all MDP components and delivers 100% success in formulation and 97.1% in execution. The single-agent version also performs strongly but slightly lags behind, while unstructured prompting results in significantly lower performance across all metrics. These results highlight the benefits of modular prompting and agent specialization in structured



(a) Pass@k performance of different models.



(b) Model performance across complexity tiers.

Figure 3: Performance comparison of language models on text-based MDP tasks. **(a)** shows Pass@1, Pass@4, and Pass@8 accuracy. **(b)** The number of problems correctly solved in each complexity tier (Low, Medium, High), with bars labeled as `solved/total`.

decision-making. For example, for an autocomplete task (Appendix C.1) where the goal is to type a prefix and select a suggestion, the action sequence generated by the single-agent model, we observe an inefficient trajectory where the agent types the full word before using the suggestion list. This approach contradicts the intended use of the interface and results in unnecessary actions, lower efficiency,

Setting	Model	State	Action	Transition	Reward	Policy
OptiAct (MA)	o3-mini	**0.850** $\pm$ 0.01	**0.919** $\pm$ 0.03	**0.837** $\pm$ 0.02	**0.866** $\pm$ 0.02	0.869 $\pm$ 0.01
OptiAct (SA)	o3-mini	0.77 $\pm$ 0.02	0.83 $\pm$ 0.05	0.72 $\pm$ 0.01	0.85 $\pm$ 0.02	**0.90** $\pm$ 0.03
Unstructured Prompting (SA)	o3-mini	0.66 $\pm$ 0.05	0.78 $\pm$ 0.01	0.67 $\pm$ 0.02	0.75 $\pm$ 0.03	0.82 $\pm$ 0.04
Unstructured Prompting (SA)	gpt-4o	0.47 $\pm$ 0.06	0.68 $\pm$ 0.03	0.51 $\pm$ 0.01	0.59 $\pm$ 0.04	0.56 $\pm$ 0.04
Unstructured Prompting (SA)	deepseek-r1	0.65 $\pm$ 0.05	0.76 $\pm$ 0.02	0.61 $\pm$ 0.01	0.80 $\pm$ 0.02	0.84 $\pm$ 0.03

Setting	Model	Successful Executions	Successful Formulations
OptiAct (MA)	o3-mini	**34/35 (97.1%)**	**35/35 (100.0%)**
OptiAct (SA)	o3-mini	23/35 (64.7%)	29/35 (85.3%)
Unstructured Prompting (SA)	o3-mini	16/35 (45.7%)	18/35 (51.4%)
Unstructured Prompting (SA)	gpt-4o	4/35 (11.4%)	5/35 (14.3.%)
Unstructured Prompting (SA)	deepseek-r1	19/35 (54.3%)	15/35 (48.6%)

Table 3: Evaluation of different models under three settings on text-based games: OptiAct with multi-agent (MA), single-agent (SA), and unstructured prompting. The top table shows average accuracy across MDP components. The bottom table reports the number of successful MDP executions and correct formulations out of 35 tasks.

and weaker reward performance. In contrast, the OptiAct agents follow a more optimal sequence by typing only the prefix and promptly selecting a suggestion, which aligns closely with the ground-truth strategy.

Sequence of actions generated by Single Agent (o3-mini) with Unstructured Prompt

```
empty_input → type_prefix → typing_prefix → type_full_word → suggestion_list → click_suggestion → selected → done
```

Sequence of actions generated by OptiAct agents (o3-mini) with Structured Prompting

```
empty_input → type_prefix → typing_prefix → suggestion_list → click_suggestion → selected → done
```

6 Failed Experiments

We initially experimented with a free-form structured format similar to the SNOP template in AhmadiTeshnizi et al. (2023), where models generate templated text for MDP components. However, the lack of strict formatting made outputs difficult to parse and often inconsistent. In contrast, our structured representation facilitates reliable integration across multiple LLM agents. While we tested smaller models (e.g., Gemini-1.5 Pro, Mistral 7B, LLaMA2-7B), they failed to consistently produce valid structured outputs and were excluded from further evaluation. Our experiments focus on verifying text-based MDP component extraction and syntactically correct code generation as a foundational step toward LLM-based decision-making. E

7 Conclusion and Future Directions

In this work, we presented a novel dataset and a pipeline to solve sequential decision making problems using a sequence of LLM agents each performing different tasks to accomplish the complex reasoning required obtain optimal policy. Automation of business decision making is an active area of research and our pipeline can be integrated into existing interfaces to create an experience for non-MDP expert to simulate different decision scenarios. In the future, we aim to refine the pipeline and enhance our dataset by incorporating infinite-horizon decision-making problems while adapting to smaller LLMs to develop a unified agent. Although current small scale LLMs are not suitable for our pipeline, finetuning them to adapt is an exciting future research direction. To reproduce this research, we will make our experiment artifacts available.

8 Limitations

One of the key limitations of our approach is the use of LLM as judge. Although this is a common trend to use LLM to perform automatic evaluation agentic AI system, the model may be biased towards the generated solution. We use different sets of models and report the variances in their evaluation. Also, as shown our approach is not suitable for small scale language models or any LLM models that do not support structured output.

References

Raha Aghaei, Ali A Kiaei, Mahnaz Boush, Javad Vahidi, Zeynab Barzegar, and Mahan Rofoosheh. 2025. The potential of large language models in supply chain management: Advancing decision-making, efficiency, and innovation. *arXiv preprint arXiv:2501.15411*.

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. 2023. Optimus: Optimization modeling using mip solvers and large language models. *arXiv preprint arXiv:2310.06116*.

Anurag Ajay, Seungwook Han, Yilun Du, Shuang Li, Abhi Gupta, Tommi S. Jaakkola, Joshua B. Tenenbaum, Leslie Pack Kaelbling, Akash Srivastava, and Pulkit Agrawal. 2023. Compositional foundation models for hierarchical planning. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Sara AlMahri, Liming Xu, and Alexandra Brintrup. 2024. Enhancing supply chain visibility with knowledge graphs and large language models. *arXiv preprint arXiv:2408.07705*.

Richard Bellman. 1957. *Dynamic Programming*. Princeton University Press.

Dimitri Bertsekas. 2012. *Dynamic programming and optimal control: Volume I*, volume 4. Athena scientific.

Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, and 1 others. 2023. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on Robot Learning*, pages 287–318. PMLR.

Ethan Brooks, Logan Walls, Richard L. Lewis, and Satinder Singh. 2023. Large Language Models Can Implement Policy Iteration. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. J. Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. *ArXiv*, abs/2005.14165.

Hailin Chen, Fangkai Jiao, Mathieu Ravaut, Nawshad Farruque, Xuan Phi Nguyen, Chengwei Qin, Manan Dey, Bosheng Ding, Caiming Xiong, Shafiq Joty, and 1 others. 2024. Structtest: Benchmarking llms' reasoning through compositional structured outputs. *arXiv preprint arXiv:2412.18011*.

Marc-Alexandre Côté, Ben Kybartas, Matthew Hausknecht, Layla El Asri, Mohammed Adada, Adam Trischler, and Kaheer Suleman. 2018. Textworld: A learning environment for text-based games. *arXiv preprint arXiv:1806.11532*.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 181 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Deborah Ferreira and André Freitas. 2020. Natural language premise selection: Finding supporting statements for mathematical text. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 2175–2182, Marseille, France. European Language Resources Association.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Noah Brown, Tomas Jackson, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. 2022. Inner monologue: Embodied reasoning through planning with language models. *Preprint*, arXiv:2207.05608.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Beibin Li, Konstantina Mellou, Bo Zhang, Jeevan Pathuri, and Ishai Menache. 2023. Large language models for supply chain optimization. *arXiv preprint arXiv:2307.03875*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. Code as policies: Language model programs for embodied control. *Preprint*, arXiv:2209.07753.

Bill Yuchen Lin, Yicheng Fu, Karina Yang, Prithviraj Ammanabrolu, Faeze Brahman, Shiyu Huang, Chandra Bhagavatula, Yejin Choi, and Xiang Ren. 2023. Swiftsage: A generative agent with fast and slow thinking for complex interactive tasks. In *Advances in Neural Information Processing Systems*.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. 2018. Reinforcement learning on web interfaces using workflow-guided exploration. *arXiv preprint arXiv:1802.08802*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*.

OpenAI. 2024. Openai unveils 'o3' reasoning ai models in test phase. Accessed: 2025-02-17.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Elena Pashenkova, Irina Rish, and Rina Dechter. 1996. Value iteration and policy iteration algorithms for markov decision problem. In *AAAI'96: Workshop on Structural Issues in Planning and Temporal Reasoning*, volume 39. Citeseer.

Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.

Yinzhu Quan and Zefang Liu. 2024. Invagent: A large language model based multi-agent system for inventory management in supply chains. *arXiv preprint arXiv:2407.11384*.

Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and 1 others. 2023. Nl4opt competition: Formulating optimization problems based on their natural language descriptions. In *NeurIPS 2022 Competition Track*, pages 189–203. PMLR.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*.

Muhammad Shakoor, Munawar Sheikh, and et al. 2016. Enhancing energy efficiency in smart grids using reinforcement learning. *Sustainable Energy, Grids and Networks*, 8:14–23.

Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. 2017. World of bits: An open-domain platform for web-based agents. In *International Conference on Machine Learning*, pages 3135–3144. PMLR.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2021. Alfworld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations (ICLR)*.

Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction (2nd ed.)*. MIT Press.

Xiangru Tang, Yiming Zong, Jason Phang, Yilun Zhao, Wangchunshu Zhou, Arman Cohan, and Mark Gerstein. 2023. Struc-bench: Are large language models really good at generating complex structured data? *arXiv preprint arXiv:2309.08963*.

Kevin Wang, Junbo Li, Neel P. Bhatt, Yihan Xi, Qiang Liu, Ufuk Topcu, and Zhangyang Wang. 2024. On the planning abilities of openai's o1 models: Feasibility, optimality, and generalizability. *Preprint*, arXiv:2409.19924.

Zihan Wang, Rui Pan, Jiarui Yao, Robert Csordas, Linjie Li, Lu Yin, Jiajun Wu, Tong Zhang, Manling Li, and Shiwei Liu. 2025. Chain-of-experts: Unlocking the communication power of mixture-of-experts models. *arXiv preprint arXiv:2506.18945*.

Nicholas R. Waytowich, Devin White, MD Sunbeam, and Vinicius G. Goecks. 2024. Atari-gpt: Benchmarking multimodal large language models as low-level policies in atari games. *Preprint*, arXiv:2408.15950.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, and 1 others. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*.

Wayne L Winston. 2004. *Operations research: applications and algorithm*. Thomson Learning, Inc.

Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Wei Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. 2025. Optibench meets resocratic: Measure and improve llms for optimization modeling. *Preprint*, arXiv:2407.09887.

Shinn Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *NeurIPS 2022*.

Shiwali Yao, Yuchen Wang, Baolin Peng, Ming Zhou, and Li Deng. 2017. Web-based interactive tasks for measuring language understanding. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Shunyu Yao, Rohan Rao, Matthew Hausknecht, and Karthik Narasimhan. 2020. Keep calm and explore: Language models for action generation in text-based games. *Preprint*, arXiv:2010.02903.

Chao Yu, Jiming Liu, Shamim Nemati, and Guosheng Yin. 2021. Reinforcement learning in healthcare: A survey. *ACM Computing Surveys (CSUR)*, 55(1):1–36.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T. Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. 2023. Metamath: Bootstrap your own mathematical questions for large language models. *arXiv preprint arXiv:2309.12284*.

Zhaocheng Zhu, Yuan Xue, Xinyun Chen, Denny Zhou, Jian Tang, Dale Schuurmans, and Hanjun Dai. 2024. Large language models can learn rules. *Preprint*, arXiv:2310.07064.

A Appendix

A.1 Structured MDP Component

We pass the structured MDP definition, as shown in Listing 4, to the prompt. The placeholder RETURN_FORMAT is substituted by this object. The LLM returns this class object (a data model) with appropriate types and values inferred from the problem description.

A.2 Dataset

We evaluate models on a diverse benchmark of text-based MDP problems spanning two primary categories: abstract decision-making domains and grounded, interactive web-based games. The first dataset (Table 4) consists of 22 problems drawn from real-world domains such as stock trading, hospital resource management, cybersecurity planning, and dynamic pricing. Each domain is formulated as a finite MDP, where states capture high-level semantic context (e.g., "Secure" vs. "Vulnerable") and actions correspond to discrete intervention choices (e.g., "Invest", "Don't Invest"). These problems vary significantly in complexity, with state and action spaces ranging from small, interpretable domains (e.g., 2×2) to large-scale tasks with over 1,000 states and actions. To quantify the complexity of each task, we compute a score based on the logarithm of the state-action space size and use it to assign tasks into three tiers: Low, Medium, and High.

Complementing this, Table 5 presents a second dataset derived from MiniWoB++-like environments, where agents must complete goal-directed tasks through textual instructions and simulated web UI interactions. These tasks involve context-dependent action choices (e.g., clicking a button, selecting a menu item) and require reasoning over visual and textual interface states. Similar to the first dataset, we compute the decision space size and assign complexity tiers based on a logarithmic scale. Notably, while the domain-level problems emphasize structured decision reasoning, the web-based games focus on grounded interaction and execution fidelity. Together, these two datasets provide a balanced and comprehensive testbed for evaluating the reasoning, planning, and generalization capabilities of large language models across symbolic and grounded decision-making tasks.

We categorize MDP problems into three tiers based on the complexity score $\mathcal{C}$: *Low* for $\mathcal{C} < 1.0$ (trivially solvable), *Medium* for $1.0 \leq \mathcal{C} < 2.5$ (moderate decision space), and *High* for $\mathcal{C} \geq 2.5$ (large and complex decision spaces).

B Evaluation Metrics

1. **Structured Output Accuracy.** Measures whether the LLM produces correct and structured format of MDP components into the predefined structured schema as shown in List 1. If a model fails to extract structured MDP component, it is treated as unsolved.

2. **Component Accuracy.** Measures the accuracy of extracting each MDP component $y \in \{\mathcal{S}, \mathcal{A}, \mathcal{T}, R\}$ by comparing predicted and ground-truth elements across all instances i. For example, if $y = \mathcal{A}$, and the model predicts 3 out of 5 actions correctly, then action accuracy becomes 0.6.

$$\text{Accuracy}_{\text{component}}(y) = \frac{|\{i \mid y_i^{\text{pred}} = y_i^{\text{true}}\}|}{|\{i\}|} \tag{2}$$

3. **Formulation Success Rate.** Measures the proportion of problems for which the model achieves at least 70% accuracy across all MDP components.

$$\text{SR}_{\text{formulation}} = \frac{\sum_{i=1}^{N_{\text{total problems}}} \mathbb{1}\left[\text{Acc}_i \geq 0.70\right]}{N_{\text{total problems}}} \tag{3}$$

4. **Execution Success Rate.** Computes the fraction of problems for which valid, executable code is generated within the allowed attempts.

$$\text{SR}_{\text{execution}} = \frac{N_{\text{execution success}}}{N_{\text{total problems}}} \tag{4}$$

5. **Pass@k.** Computes the fraction of problems solved within k attempts out of the total num-

Table 4: Text-Based MDP Problem Complexity (Sorted by Score)

/

Domain	State Description	Action Description	State Size	Action Size	Score	Tier
Stock Trading	2 states (Bull, Bear)	2 actions (Buy, Sell)	2	2	0.95	Low
Car Financing	2 states (Owned, Not Owned)	2 actions (Buy, Don't Buy)	2	2	0.95	Low
Cybersecurity Resilience	2 states (Secure, Vulnerable)	2 actions (Invest, Don't Invest)	2	2	0.95	Low
Project Selection	3 states (Low, Medium, High)	2 actions (Accept, Reject)	3	2	1.08	Medium
Machine Condition	3 states (Good, Fair, Poor)	2 actions (Maintain, Don't Maintain)	3	2	1.08	Medium
Restaurant Recommendation	2 states (Hungry, Satisfied)	3 actions (Fast Food, Casual, Fine Dine)	2	3	1.08	Medium
Machine Maintenance	2 states (Running, Broken)	4 actions	2	4	1.18	Medium
Fisheries Management	4 states (Low, Mid, High, Overfished)	2 actions (Fish, Don't Fish)	4	2	1.18	Medium
Customer Loyalty	4 states (New, Regular, Premium, Churned)	2 actions (Reward, Don't Reward)	4	2	1.18	Medium
Aircraft Engine	3 states (Good, Critical, Warning)	3 actions (Continue, Monitor, Replace)	3	3	1.20	Medium
Investment Risk Management	3 states (Low, Mid, High)	3 actions (Conservative, Balanced, Aggressive)	3	3	1.20	Medium
Cybersecurity Allocation	3 states (Low, Mid, High)	3 actions (Minimal, Standard, Enhanced)	3	3	1.20	Medium
General MDP	5 states (State 1 to 5)	2 actions (Action 1 and 2)	5	2	1.26	Medium
Printing Press Control	6 states (Idle, Setup, Running, Maintenance, Repair, Shutdown)	2 actions (Start, Stop)	6	2	1.32	Medium
Risk Management Game	11 states (Levels + End)	2 actions (Continue, Stop)	11	2	1.56	Medium
Gambling (fixed)	7 states (capital 0–6)	5 actions (bet $0–$4)	7	5	1.68	Medium
Hospital Bed Management	available beds: 0–20, days left: 0–14	11 actions (admit 0–10)	315	11	3.58	High
Inventory Management	Variable (0 to max capacity)	0 to 50	101	51	3.72	High
Gambling (5 rounds)	Exponential (game_number, wealth)	0 to wealth	160	32	3.73	High
Hospital Bed Allocation	100 states (1–100)	0 to capacity	100	101	4.01	High
Dynamic Pricing	1001 states ($0–$1000)	1001 actions	1001	1001	6.00	High

Table 5: Complexity of Text-Based Game MDPs

Problem ID	State Size	Action Size	$\|S\| \times \|A\|$	**Complexity**	**Tier**
click_next	2	1	2	0.4771	Low
click_pie_nodelay	2	1	2	0.4771	Low
colored_swatch	2	1	2	0.4771	Low
small_box	3	1	3	0.6021	Low
tic_tac_toe	3	1	3	0.6021	Low
button_click	2	2	4	0.6990	Low
click_collapsible_nodelay_1	3	2	6	0.8451	Low
click_collapsible_nodelay_2	3	2	6	0.8451	Low
menu_click	3	2	6	0.8451	Low
search_cat	3	2	6	0.8451	Low
type_hello_world	3	2	6	0.8451	Low
use_autocomplete	4	2	8	0.9542	Low
click_test_transfer	3	3	9	1.0000	Medium
checkbox	4	3	12	1.1139	Medium
click_tab	4	3	12	1.1139	Medium
login_form	5	3	15	1.2041	Medium
tw_game_10	4	4	16	1.2304	Medium
tw_game_11	4	4	16	1.2304	Medium
tw_game_12	4	4	16	1.2304	Medium
tw_game_3	4	4	16	1.2304	Medium
tw_game_4	4	4	16	1.2304	Medium
tw_game_5	4	4	16	1.2304	Medium
tw_game_6	4	4	16	1.2304	Medium
tw_game_7	4	4	16	1.2304	Medium
tw_game_8	4	4	16	1.2304	Medium
tw_game_9	4	4	16	1.2304	Medium
choose_date	5	4	20	1.3222	Medium
choose_date_nodelay_1	5	4	20	1.3222	Medium
date_picker	5	4	20	1.3222	Medium
flight_booking_nodelay_1	5	4	20	1.3222	Medium
tw_game_2	4	5	20	1.3222	Medium
choose_date_medium	6	5	30	1.4914	Medium
choose_date_nodelay_2	7	5	35	1.5563	Medium
tw_game_1	6	6	36	1.5682	Medium
flight_booking_nodelay_2	10	4	40	1.6128	Medium

ber of problems.

$$\text{Pass@k} = \frac{\text{Problems solved@k attempts}}{N_{\text{total problems}}} \tag{5}$$

6. **Average Attempts.** Calculates the average number of attempts made per problem, capped at 8.

$$\text{Avg Attempts} = \frac{1}{N_{\text{total}}} \sum_{i=1}^{N_{\text{total}}} \min(\text{Attempts}_i, 8) \tag{6}$$

B.1 Prompts

Our framework follows a structured Chain-of-Thought prompting paradigm, where complex decision-making problems are decomposed into intermediate reasoning tasks (e.g., MDP formulation, solution derivation, code validation) handled by specialized agents. This enables explicit step-by-step reasoning and allows verification and modular debugging (Kojima et al., 2022; Chen et al., 2024).

1. Strcutured MDP component extraction prompt: In Listing 3 the structured prompt used for zero-shot extraction of MDP components from natural language descriptions. It guides the model through explicit reasoning steps to identify states, actions, transitions, rewards, and discount factors using a Chain-of-Thought formulation.

2. Algorithm Formulation Prompt: Using the prompt in Listing 5, we compute the pseudocode of value iteration algorithm for the given problem. The resulting value function is used to derive the optimal policy by selecting the action that maximizes expected return at each state.

3. Code generation: We translate the given pseudocode into executable Python code using prompt in Listing 6 that solves the MDP and writes the resulting policy and value function to `output.json`.

4. Code fix: If previously generated code is error prone and not executable, we fix it by using prompt in Listing 7 until the maximum number of attempts are reached.

5. Evaluation: To evaluate the correctness of predicted MDP components, we compare the annotated ground truth solution vs the predicted solution. For example, in the case of transitions, the LLM is instructed to compare the predicted transition structure with the ground truth in a relaxed manner—assessing syntactic and semantic similarity. This enables more flexible and human-aligned scoring, especially when state or action names differ slightly but convey the same intent. The LLM returns a soft accuracy score between 0 and 1, where higher values indicate closer alignment with the true component. Similar prompts are used to evaluate states, actions, rewards, and policies, each with domain-specific guidelines for what constitutes a correct match (from Listing 8 to 12). This method allows us to quantify model performance on structured outputs using judgment-based supervision, without relying on brittle token-level exact match metrics.

C Experiments

C.1 Discussion

Now, we discuss in detail how OptiAct agents compare with unstructured prompt agents with an example. As shown in the problem description box, the goal of the autocomplete task is to efficiently complete a word by selecting a suggestion rather than typing it out in full. This ideal behavior involves typing only a prefix to trigger the suggestion list, followed by selecting the correct option to earn a reward and minimize effort. In the action sequence generated by the single-agent model, we observe an inefficient trajectory where the agent types the full word before using the suggestion list (see box titled *Sequence of actions generated by Single Agent (o3-mini) with Unstructured Prompt*). This approach contradicts the intended use of the interface and results in unnecessary actions, lower efficiency, and weaker reward performance. In contrast, the OptiAct agents follow a more optimal sequence—typing only the prefix and promptly selecting a suggestion—which aligns closely with the ground-truth strategy (see box titled *Sequence of actions generated by OptiAct agents (o3-mini) with Structured Prompting*). This structured policy not only mirrors human-like behavior but also demonstrates superior state-action abstraction enabled by OptiAct's modular prompting design. These results highlight the effectiveness of structured prompting in guiding large language models toward intelligent, goal-aligned decision-making in interactive environments.

Listing 2: MDP Checker Prompt with Few-Shot Examples

```
You are an MDP checking agent. Given a natural language PROBLEM_STATEMENT,
determine whether it contains the required MDP components:
States, Actions, Transitions, Rewards, Time Horizon, Discount Factor.

For each component, label as one of: Present / Inferred / Missing

Return in the following format:
States: ...
Actions: ...
Transitions: ...
Rewards: ...
Time Horizon: ...
Discount Factor: ...
Suitability: Yes / No

### Example 1
"A robot moves in a 5x5 grid. It can move up, down, left, or right. It receives
    ↪ +1
on reaching the goal. The task ends after 10 steps."


### Example 2
"A user browses a website. Sometimes they click links or leave."

Return:
States: Missing
Actions: Missing
Transitions: Missing
Rewards: Missing
Time Horizon: Missing
Discount Factor: Missing
Suitability: No

# Problem Statement
{PROBLEM_STATEMENT}

# Return Format
{RETURN_FORMAT}
```

Listing 3: Structured MDP component extraction prompt

```
    """You are an expert AI bot specializing in solving Markov Decision Process
        ↪ (MDP) problems. Your task is to formulate and solve the given problem
        ↪ using the MDP framework. Follow the instructions carefully and ensure
        ↪ your response is precise, logical, and complete.

    **Instructions:**
    1. **Understand the Problem:** Analyze the provided problem description
        ↪ thoroughly. Identify the key elements that can be mapped to the MDP
        ↪ framework.
    2. **Define MDP Components:** Explicitly define the following components for
        ↪  the problem:
    - **States:** A finite set of states representing all possible situations
        ↪ the system can be in.
    - **Actions:** A finite set of actions available to the agent in each state.
    - **Transition Function:** The probabilities of transitioning from one state
        ↪  to another, given the current state and action.
    - **Reward Function:** The rewards associated with state transitions or
        ↪ state-action pairs.
    - **Discount Factor:** A value between 0 and 1 that determines the present
        ↪ value of future rewards.

    ### Return Format:
    {RETURN_FORMAT}
```

Listing 4: Definition of structured MDP component used as RETURN_FORMAT in the prompt. Component extraction agent returns this object with associated values extracted from the text description.

```
title: MDPComponents
$defs:
  Reward:
    title: Reward
    type: object
    properties:
      current_state:
        title: Current State
        description: The state the agent is in.
        type: string
      action:
        title: Action
        description: The action taken by the agent.
        type: string
      next_state:
        title: Next State
        description: The state the agent transitions to.
        type: string
      reward:
        title: Reward
        description: The reward received by the agent.
        type: string
    required:
      - current_state
      - action
      - next_state
      - reward
  Transition:
    title: Transition
    type: object
    properties:
      current_state:
        title: Current State
        description: The state the agent is in.
        type: string
      action:
        title: Action
        description: The action taken by the agent.
        type: string
      next_state:
        title: Next State
        description: The state the agent transitions to.
        type: string
      probability:
        title: Probability
        description: The probability of transitioning to the next state given
            ↪ the current state and the action taken.
        anyOf:
          - type: number
          - type: integer
    required:
      - current_state
      - action
      - next_state
      - probability
```

```yaml
properties:
  reason:
    title: Reason
    description: Explanation of how the problem fits into the MDP framework
    type: string
  states:
    title: States
    description: A finite set of states the system can be in. A list of strings.
    type: array
    items:
      type: string
  actions:
    title: Actions
    description: A finite set of actions the agent can take. A list of strings.
    type: array
    items:
      type: string
  transitions:
    title: Transitions
    description: The probabilities of moving from one state to another given the
      ↪  current state and the action taken. A list of dictionaries.
    type: array
    items:
      $ref: "#/$defs/Transition"
  rewards:
    title: Rewards
    description: The rewards received by the agent after performing certain
      ↪ actions in certain states. A list of dictionaries.
    type: array
    items:
      $ref: "#/$defs/Reward"
  discount_factor:
    title: Discount Factor
    description: A discount factor between 0 and 1 that determines the present
      ↪ value of future rewards.
    anyOf:
      - type: number
      - type: integer
    default: 1.0
required:
  - reason
  - states
  - actions
  - transitions
  - rewards
```

Listing 5: Prompt to formulate Value Iteration Algorithm

```
You are an expert AI bot specializing in solving Markov Decision Process (MDP) problems. Your task is to formulate and solve the given problem using the MDP framework. Follow the instructions carefully and ensure your response is precise, logical, and complete.

**Instructions:**
**Problem Formulation**
1. **Understand the Problem:** Analyze the provided problem description thoroughly. Identify the key elements that can be mapped to the MDP framework.
2. **Define MDP Components:** Explicitly define the following components for the problem:
- **States:** A finite set of states representing all possible situations the system can be in.
- **Actions:** A finite set of actions available to the agent in each state.
- **Transition Function:** The probabilities of transitioning from one state to another, given the current state and action.
- **Reward Function:** The rewards associated with state transitions or state-action pairs.
- **Discount Factor:** A value between 0 and 1 that determines the present value of future rewards.
3. **Formulate the Value Function:** Construct the Bellman equation to represent the value function for the problem. Ensure it is mathematically precise and aligns with the problem's objective.

**Algorithm: Value Iteration Algorithm (Adapted from Sutton & Barto, 2018)**
Initialize V(s) arbitrarily for all s     S
Repeat until convergence:
    For each state s     S:
        V(s)     max_a   _ {s'} P(s' | s, a) [ R(s, a, s') +    V(s') ]
Return the final value function V and the derived optimal policy   (s) = argmax_a   _ {s'} P(s' | s, a) [ R(s, a, s') +    V(s') ]

4. **Specify Constraints:** List all relevant constraints that must be considered in the problem. Ensure they are non-redundant and directly applicable to the MDP formulation.
5. **Validate Coherence:** Verify that your formulation is logically consistent, free of errors, and aligns with the problem's requirements .
6. **Provide Explanation:** Clearly explain your formulation, including the reasoning behind each component. Use diagrams, flowcharts, or pseudo-code if necessary to enhance understanding.
**Problem Information:**
{PROBLEM_INFO}
```

Listing 6: Code generation prompt from pseudocode.

```

You are an expert in coding. Generate a Python program that implements the value iteration algorithm described in the given formulation section. The final program should save the output in a file named "output.json".

When writing the code, adhere to the following requirements:
- ONLY generate the code. Do not include any extra explanation or output text.
- Wrap the code using markdown-style triple backticks (```).
- Follow Python PEP 8 style for clean and readable code.
- Assume all necessary libraries are already installed.

### INSTRUCTIONS:
1. **Read Input**: Use the MDP components and pseudocode provided in the formulation.
2. **Initialization**: Initialize the MDP model based on the provided state, action, transition, and reward structures.
3. **Parameters**: If unspecified, assume the discount factor \( \gamma \) and maximum number of iterations to be 0.99 and 10000, respectively.
4. **Algorithm Implementation**: Translate the pseudocode logic step-by-step into working Python code. Implement Bellman updates iteratively until convergence or until the maximum iteration count is reached.
5. **Output Format**: After solving, store the policy \( \pi \) (as a list of optimal actions per state) and the value function \( V \) (as a list of floats) into a JSON file named "output.json", indented with 4 spaces.
6. **Libraries**: Do not install any libraries. You may use built-in Python and NumPy only.
7. **Markdown**: Wrap the entire code output using triple backticks for proper formatting.

Take a deep breath and generate the final code with attention to structure, correctness, and formatting.

## Formulation:
{PROBLEM_FORMULATION}

### Return Format:
{RETURN_FORMAT}
```

Listing 7: Code fix prompt.

```
Your task is to fix the Python code that was previously generated to solve a Markov Decision Process (MDP) problem. The current code has encountered an error, and you need to correct it.

    ### INSTRUCTIONS:
    1. **Analyze the Error**: Carefully read the error message provided and understand the issue in the code.
    2. **Correct the Code**: Modify the code to fix the error while ensuring it adheres to the original problem formulation.
    3. **Maintain Structure**: Ensure that the code is well-structured, following Python PEP 8 style guidelines.
    4. **No Extra Libraries**: Do not introduce any new third-party libraries; use only standard Python libraries and NumPy for computation.
    5. **Output File**: Ensure the code correctly saves the results in "output.json" with indents of 4 spaces.
    6. **Preserve Logic**: Keep the original logic intact as much as possible, only adjusting what is necessary to resolve the error.
    7. **Code Format**: Return only the corrected code wrapped in markdown triple backticks (```), without any additional text or explanations.

    ### Formulation:
    {PROBLEM_FORMULATION}

    ### Previous Code:
    {PREVIOUS_CODE}

    ### Error:
    {CODE_ERROR}

    Carefully review the provided details and make the necessary corrections to ensure the code functions as intended.
```

Listing 8: Action evaluation prompt.

```
You are given a Markov Decision Process (MDP) problem. Your role is to evaluate whether given true actions and predicted actions of the problem are correct or not. Provide a score between 0 and 1. Instructions for evaluation: Don't be strict too match all actions name exactly. Just check if the predicted actions are similar to the true actions. If the predicted actions are similar to the true actions, then give a score close to 1. If the predicted actions are not similar to the true actions, then give a score close to 0.
...

Problem: {problem_statement}
True Actions: {true_actions}
Predicted Actions: {predicted_actions}

Response Instructions:
{RETURN_FORMAT}
```

Listing 9: State evaluation prompt.

```
You are given a Markov Decision Process (MDP) problem. Your role is to evaluate whether given true states and predicted states of the problem are correct or not. Provide a score between 0 and 1. Instructions for evaluation: Don't be strict too match all states name exactly. Just check if the predicted states are similar to the true states and are equal. If the predicted states are similar to the true states, then give a score close to 1. If the predicted states are not similar to the true states, then give a score close to 0.

Problem: {problem_statement}
True States: {true_states}
Predicted States: {predicted_states}

Response Instructions:
{RETURN_FORMAT}
```

Listing 10: Reward evaluation prompt

```

 You are given a Markov Decision Process (MDP) problem. Your role is to evaluate whether given true rewards and predicted rewards of the problem are correct or not. Provide a score between 0 and 1. Instructions for evaluation: Don't be strict too match all rewards state name exactly. Just check if the predicted rewards states are similar to the true rewards states. If the predicted rewards are similar to the true rewards, then give a score close to 1. If the predicted rewards are not similar to the true rewards, then give a score close to 0.

Problem: {problem_statement}
True Rewards: {true_rewards}
Predicted Rewards: {predicted_rewards}

Response Instructions:
{RETURN_FORMAT}
```

Listing 11: Transition evaluation prompt.

```
You are given a Markov Decision Process (MDP) problem. Your role is to evaluate whether given true transitions and predicted transitions of the problem are correct or not. Provide a score between 0 and 1. Instructions for evaluation: Don't be strict too match all transitions state name exactly. Just check if the predicted transitions states are similar to the true transitions states. If the predicted transitions are similar to the true transitions, then give a score close to 1. If the predicted transitions are not similar to the true transitions, then give a score close to 0.
...

Problem: {problem_statement}
True Transitions: {true_transitions}
Predicted Transitions: {predicted_transitions}

Response Instructions:
{RETURN_FORMAT}
```

Listing 12: Policy evaluation prompt.

```
You are given a Markov Decision Process (MDP) problem. Your role is to evaluate whether given true policy of action sequences and predicted policy of the problem are correct or not. Provide a score between 0 and 1.

Problem: {problem_statement}
True States: {true_policy}
Predicted States: {predicted_policy}

Response:
{{
"score": between 0 and 1,
"reason": Reason for the score
}}
```

Problem Description: Autocomplete Task

You are on a webpage with an autocomplete input field that provides intelligent suggestions as you type. The input field starts completely empty, and your goal is to successfully use the autocomplete feature to select the correct suggestion rather than typing out the entire word manually. The autocomplete process begins with an empty input field, where the user starts typing a prefix, prompting the system to generate matching suggestions in a dropdown list after sufficient input; to complete the task efficiently, the user should click on the correct suggestion rather than typing the entire word, which earns a reward of +1 and reflects proper use of the interface.
Success is defined as selecting a suggestion rather than typing the entire word.

Sequence of actions generated by Single Agent (o3-mini) with Unstructured Prompt

```
empty_input → type_prefix → typing_prefix → type_full_word → suggestion_list → click_suggestion → selected → done
```

Sequence of actions generated by OptiAct agents (o3-mini) with Structured Prompting

```
empty_input → type_prefix → typing_prefix → suggestion_list → click_suggestion → selected → done
```

The single-agent model exhibits inefficient behavior by redundantly typing the full word before interacting with the suggestion list—thereby undermining the purpose of autocomplete—OptiAct agents follow a more optimal sequence by typing only a partial prefix and immediately selecting from the suggestions. This behavior closely mirrors the intended human-like interaction pattern and adheres to the ground-truth policy, resulting in faster task completion and higher reward.

D LLM Usage

When preparing this manuscript, we utilized a Large Language Model (LLM) to assist with various aspects of the writing and research process. The LLM was employed for several key tasks: The LLM helped improve sentence structure, grammar, and overall readability of the manuscript, ensuring clear and professional academic writing throughout the paper.