Prompt Optimization with Large Reasoning Models for Event Extraction

Anonymous ACL submission

Abstract

Large Reasoning Models (LRMs) such as DeepSeek-R1 and o1 have led to claims that prompt optimization is no longer necessary. We test this on end-to-end event extraction (EE), presenting the first application of automated prompt optimization to this challenging structured prediction task. Using an MCTS-based framework, we evaluate four models, two LRMs (DeepSeek-R1, o1) and two LLMs (GPT-4o, GPT-4.5), as both task models and optimizers, and additionally validate on two newer LRMs (o3, o4-mini). All models benefit substantially (up to +27.75 AC F1), with optimized prompts automatically discovering extraction rules absent from human-written guidelines. LRMs prove particularly effective as optimizers, producing shorter, rule-oriented prompts that converge faster with lower variance (non-overlapping 95% bootstrap CIs). Integrating batch prompting further reduces cost while improving accuracy. These findings generalize across symbolic reasoning, biomedical NER, and open-ended planning, establishing that prompt optimization remains essential in the era of LRMs.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities across diverse NLP tasks, yet their proficiency in complex reasoning has often been limited (Zhou et al., 2022). To address this, a new class of Large Reasoning Models (LRMs) has emerged, trained with reinforcement learning over chain-of-thought to substantially improve multi-step reasoning. Prominent examples include DeepSeek-R1 (Guo et al., 2025) and OpenAI's o1 (Zhong et al., 2024), o3, and o4-mini, which have set new standards on mathematical, coding, and scientific benchmarks.

Prompt optimization, the process of automatically refining input prompts to improve model outputs without modifying model weights (Zhou et al., 2022; Yang et al., 2024; Li et al., 2025), has been an active area of research for LLMs, with methods such as PromptAgent (Wang et al., 2024b), OPRO (Yang et al., 2024), TextGrad (Yuksekgonul et al., 2024), MIPROv2 (Opsahl-Ong et al., 2024), and GEPA (Khattab et al., 2026) demonstrating substantial gains on classification and reasoning benchmarks. However, these methods have not been applied to complex structured prediction tasks such as event extraction (EE), where models must navigate intricate schema constraints and even powerful LLMs continue to struggle (Gao et al., 2024; Srivastava et al., 2023). Meanwhile, the emergence of LRMs has led to a widespread belief, echoed in official model documentation (OpenAI, 2025; Mantaras, 2025) and community discussions (Wang et al., 2024a; Together AI, 2025; Menendez et al., 2025), that these models may no longer require prompt optimization at all. *No prior work has applied automated prompt optimization to EE, nor has any work studied whether LRMs benefit from or can serve as effective prompt optimizers.*

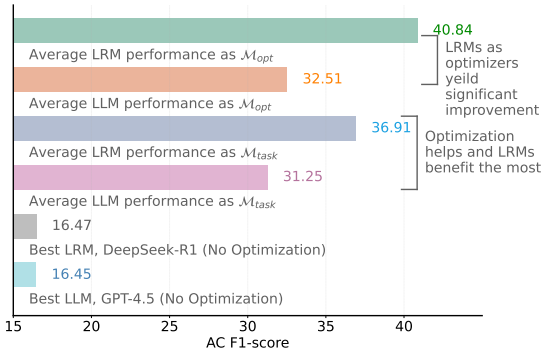


Figure 1: Summary of our main results, where LRMs and LLMs are used as either the task model ($\mathcal{M}_{task}$) or the optimizer ($\mathcal{M}_{opt}$) in prompt optimization. We observe a strong advantage of LRMs over LLMs in both roles.

To address both gaps, we present the first application of automated prompt optimization to end-

to-end EE, a challenging structured prediction task that requires identifying event triggers, classifying their types, and extracting argument spans with their semantic roles (Doddington et al., 2004). Recent LLM-based EE approaches include code-style prompting (Sainz et al., 2024; Li et al., 2024; Wang et al., 2023), instruction tuning (Srivastava et al., 2025b), and reasoning-augmented extraction (Parekh et al., 2025). While Srivastava et al. (2025b) showed that annotation guidelines, including machine-generated ones, substantially improve EE models, these studies compare fixed prompt variants rather than automatically optimizing the prompt itself. Our framework fills this gap, iteratively refining task instructions and event guidelines based on model feedback using MCTS (Wang et al., 2024b). We evaluate four models, two LRMs (DeepSeek-R1, o1) and two LLMs (GPT-4o, GPT-4.5), as both task models and prompt optimizers, with additional validation on two newer LRMs (o3, o4-mini), and further test generalizability on Geometric Shapes (Suzgun et al., 2022), NCBI Disease NER (Doğan et al., 2014), and Gridworld Planning (Aghzal et al., 2024).

Our main findings (summarized in Fig. 1) are:

Prompt optimization dramatically improves EE performance. All models, both LLMs and LRMs, benefit substantially from automated prompt optimization, with gains of up to +27.75 AC F1. The optimized prompts automatically discover domain-specific extraction rules absent from the original human-written guidelines, reducing common errors such as implicit trigger over-generation and argument span drift.

LRMs are surprisingly strong, both as task models and as optimizers. While all models benefit from prompt optimization, LRMs tend to reach higher post-optimization performance and, in many settings, show larger gains, particularly DeepSeek-R1, which achieves the highest scores in nearly every configuration. More strikingly, when used as *optimizers*, LRMs produce shorter, more rule-oriented prompts, converge faster in MCTS (peaking at depth 2–3 vs. depth 4–5 for LLMs) with lower variance.

Batch prompting improves the optimization loop. Integrating batch prompting (Cheng et al., 2023) into the MCTS feedback loop not only reduces inference cost but also improves task accuracy compared to single-instance querying, a practical methodological contribution for efficient prompt search.

Findings generalize across tasks. The same qualitative patterns hold across symbolic reasoning, biomedical NER, and open-ended planning: prompt optimization consistently improves all models, with LRMs achieving the highest post-optimization performance.

2 Related Works

LLM-based Event Extraction. EE requires identifying event triggers and classifying their types, as well as extracting arguments with their semantic roles (Doddington et al., 2004). Recent LLM-based approaches include *in-context learning* with code-style prompting (Wang et al., 2023; Li et al., 2023, 2024; Sainz et al., 2024) and *supervised fine-tuning* with annotation guidelines (Srivastava et al., 2025b; Gao et al., 2024), as well as reasoning-augmented extraction (Parekh et al., 2025). Notably, Srivastava et al. (2025b) showed that annotation guidelines, including machine-generated ones, substantially improve instruction-tuned EE models, though their work compares fixed prompt variants rather than automatically optimizing prompts. Despite these advances, even powerful LLMs continue to struggle with EE (Gao et al., 2024; Wei et al., 2024). Our work is the first to apply automated prompt optimization to EE, iteratively refining guidelines from model feedback.

Prompt Optimization and Large Reasoning Models. Prompt optimization adapts LLMs to downstream tasks without modifying model weights, with approaches ranging from gradient-based search (Li and Liang, 2021; Shin et al., 2020) to black-box techniques that refine prompts via perturbation, evolutionary operations, or LLM-driven rewriting (Zhou et al., 2022; Pryzant et al., 2023; Guo et al., 2024; Srivastava et al., 2024; Wang et al., 2024b), and recent program-level systems such as TextGrad (Yuksekgonul et al., 2024), MIPROv2 (Opsahl-Ong et al., 2024), and GEPA (Khattab et al., 2026). However, these methods have been evaluated on classification and reasoning benchmarks, not on structured prediction tasks like EE. Meanwhile, LRMs such as DeepSeek-R1 (Guo et al., 2025), OpenAI o1 (Zhong et al., 2024), o3, o4-mini, and QwQ-32B (Qwen Team, 2025), trained with reinforcement learning (RL) to reason internally, have led to

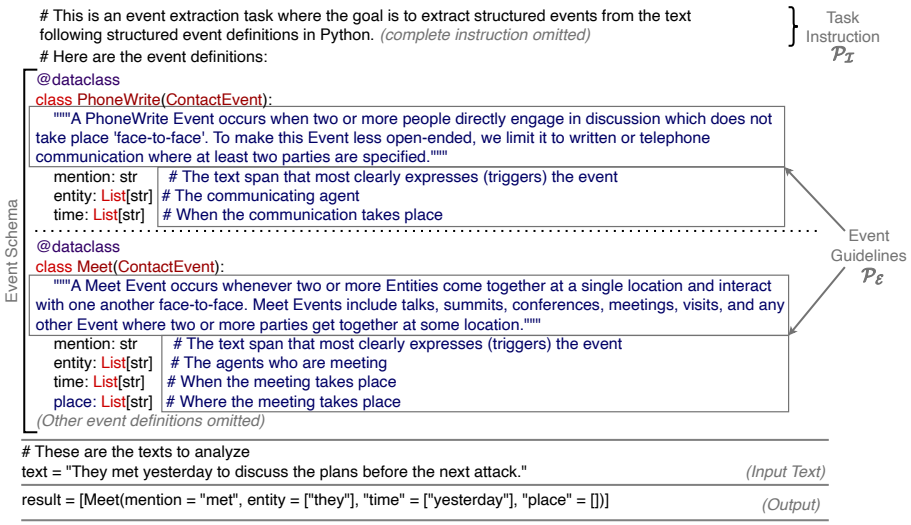


Figure 2: An example prompt for end-to-end EE used in our experiments, consisting of a task instruction and an event schema. The event schema contains information about the labels that are represented as Python classes and event guidelines defining both the event classes and the arguments. In prompt optimization, we refine both the task instruction and event guidelines (shown for two events; others omitted due to space limits) to generate more effective prompts for the task model.

claims that prompt engineering is no longer necessary (OpenAI, 2025; Mantaras, 2025; Wang et al., 2024a). Yet these claims concern *manual* prompt engineering; no prior work has studied *automated* optimization for LRMs. We fill this gap and additionally integrate batch prompting (Cheng et al., 2023) into the MCTS optimization loop, showing that it reduces cost while improving accuracy.

3 Methodology

3.1 Problem Setup

Following prior work (Wang et al., 2023; Sainz et al., 2024; Li et al., 2024; Srivastava et al., 2025b), we adopt a Python code-based representation for both input and output of the task model $\mathcal{M}_{task}$. As shown in Fig. 2, the initial prompt $\mathcal{P}_0 = [\mathcal{P}_\mathcal{I} \| \mathcal{P}_\mathcal{E}]$ consists of a **task instruction** $\mathcal{P}_\mathcal{I}$ specifying the task and output format, and **event guidelines** $\mathcal{P}_\mathcal{E}$ providing human-written descriptions of event types and argument roles annotated as Python class docstrings. Given a training set $\mathcal{D}_{train} = \{(Q_i, A_i)\}_{i=1}^{N}$, the goal is to find an optimal prompt $\mathcal{P}^*$ that maximizes a task-specific evaluation function $\mathcal{R}$ (e.g., F1 score for EE). We employ a separate optimizer model $\mathcal{M}_{opt}$ to refine both $\mathcal{P}_\mathcal{I}$ and $\mathcal{P}_\mathcal{E}$ while keeping the event schema (class definitions and argument roles) fixed.

3.2 Prompt Optimization Framework

We frame prompt optimization as a discrete search over the natural-language prompt space $\mathcal{S}$ and adopt Monte Carlo Tree Search (MCTS) following Wang et al. (2024b), modeling the process as a Markov Decision Process (MDP) where each state s_t corresponds to a prompt $\mathcal{P}_t$ and actions correspond to error-driven prompt edits.

As illustrated in Fig. 3, each MCTS node holds a prompt $\mathcal{P}_t$ and a batch of queries Q_{batch} from $\mathcal{D}_{train}$. The optimization proceeds in four steps. In Step 1, $\mathcal{M}_{task}$ generates answers for queries in Q_{batch}. The incorrect outputs are then extracted and passed through a Python interpreter to identify parsing errors, missing event types, and invalid spans (Step 2). In Step 3, we prompt $\mathcal{M}_{opt}$ with a meta-prompt m_{fb} to analyze the errors and generate structured feedback f_t, pinpointing unclear definitions and proposing fixes. In Step 4, $\mathcal{M}_{opt}$ produces the updated prompt $\mathcal{P}_{t+1} \sim p_{\mathcal{M}_{opt}}(s_{t+1} \mid s_t, f_t, m_{opt})$ in a single pass, incorporating prompt history to discourage redundant edits. For tasks involving class categories (e.g., event types in EE, entity types in NER), only categories involved in the error batch are updated; others are inherited unchanged.

Each prompt is evaluated using a reward $r_t = \mathcal{R}(s_t, f_t)$ computed as the averaged F1 across EE subtasks (TI, TC, AI, AC; defined in Section 4.1) on a held-out development set. The best prompt across the search tree is selected based on dev-set performance. The full algorithm, hyperparameter settings, and meta-prompt templates (m_{fb}, m_{opt}) are provided in Appendix A.

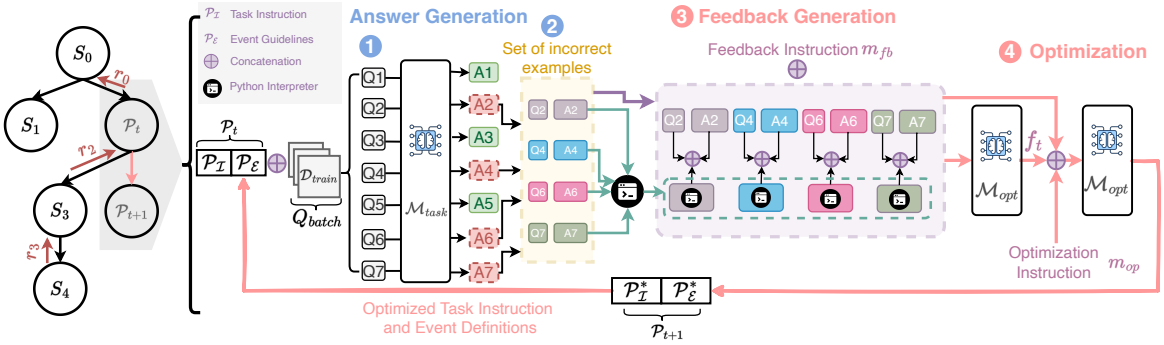


Figure 3: Overview of our prompt optimization framework. At each iteration, $\mathcal{M}_{task}$ generates outputs while $\mathcal{M}_{opt}$ analyzes errors and updates the prompt, including task instructions and event guidelines. This process continues over batches of training samples $\mathcal{D}_{train}$, and the final optimized prompt is evaluated on the development set to determine the node reward r_t.

4 Experiments

4.1 Experimental Setup

Tasks and Datasets. We experiment on the ACE05 dataset (Doddington et al., 2004), preprocessed by Huang et al. (2024). The full ACE05 schema contains 33 event types, but including all of them produces very long prompts that cause models to generate incorrect guideline updates, miss event types, and produce incomplete outputs, even within their stated context limits. To cleanly evaluate prompt optimization without these long-context confounds, we select a representative subset of 10 event types spanning diverse categories (selection criteria in Appendix A). We construct two training splits to study prompt optimization under different data regimes: **ACE**$_{low}$ (15 samples, 1 per event type plus non-event instances) tests whether prompt optimization is effective with minimal labeled data, while **ACE**$_{med}$ (120 samples, 10 per type) examines whether gains scale with more training examples. We use a 100-example development set and a 250-example test set sampled from ACE05. Statistics are in Table 5 (Appendix A). For generalization (Section 5), we include Geometric Shapes (Suzgun et al., 2022) (symbolic reasoning), NCBI Disease NER (Doğan et al., 2014) (biomedical NER), and Gridworld Planning (Aghzal et al., 2024) (open-ended planning).

Evaluation. Following Huang et al. (2024), we evaluate EE with four F1 metrics: Trigger Identification (TI), Trigger Classification (TC), Argument Identification (AI), and Argument Classification (AC). AC is the strictest end-to-end metric, requiring correct triggers, arguments, and role labels; we primarily report AC in our discussion. We also experimented with alternative reward configurations (e.g., AC-only, weighted combinations) and found that equally averaging all four subtask F1 scores yields the best AC performance; using AC alone as the reward degrades upstream subtasks, causing cascading failures (ablation in Appendix B.2). Full results across all metrics are in Appendix C. For Geometric Shapes, we report accuracy; for NCBI Disease NER, micro-F1 on strict spans; for Gridworld Planning, exact-match accuracy.

Experimental Settings and Baselines. We evaluate two LRMs (DeepSeek-R1, o1) and two LLMs (GPT-4.5, GPT-4o) as both $\mathcal{M}_{opt}$ and $\mathcal{M}_{task}$; we additionally evaluate o3 and o4-mini to test whether our findings hold on newer LRMs (setup in Appendix A.6). We first run **shallow MCTS (depth 1)** on ACE$_{low}$ and ACE$_{med}$, then **full MCTS (depth 5)** on ACE$_{med}$, expanding three children per node. We employ batch prompting (Cheng et al., 2023) when querying $\mathcal{M}_{task}$ (Step 1 in Fig. 3), which also improved accuracy over single-instance querying. DeepSeek-R1 is deployed locally and quantized to 2.5 bits using UnSloth (Daniel Han and team, 2023), which shows minimal degradation even at lower precisions. We report 95% bootstrap confidence intervals for key comparisons to ensure statistical robustness. Additional details are in Appendix A.

4.2 Experimental Results

Our main results are presented in Table 1. We discuss the following research questions (RQs).

RQ1: Do LRMs benefit from prompt optimization in EE? As shown in Table 1, all four models struggle on zero-shot EE, scoring between 12 and 17 AC F1, confirming that end-to-end extraction remains challenging even for frontier models. A sin-

$\mathcal{M}_{task}$	Optimizer LLMs/LRMs ($\mathcal{M}_{opt}$)					#Output
	No Opt.	**GPT-4o**	**GPT-4.5**	**o1**	**DS-R1**	**Tokens**
MCTS at depth 1 trained on ACE_{low} (Development Set)						
GPT-4o	12.68	18.18 +5.50	16.67 +3.99	18.83 +6.15	**20.15** +7.47	15.31
GPT-4.5	16.47	19.33 +2.86	16.47 00.00	19.32 +2.85	**22.31** +5.84	24.57
o1	13.94	18.96 +5.02	18.57 +4.63	20.29 +6.35	**21.92** +7.98	489.67
DS-R1	16.45	18.67 +2.22	18.57 +2.12	21.83 +5.38	**24.66** +8.21	217.71
MCTS at depth 1 trained on ACE_{med} (Development Set)						
GPT-4o	12.68	22.32 +9.64	**27.54** +14.86	26.30 +13.62	25.10 +12.42	17.31
GPT-4.5	16.47	29.63 +13.16	35.94 +19.47	**36.51** +20.04	35.42 +18.95	28.75
o1	13.94	30.19 +16.25	36.67 +22.73	**36.98** +23.04	36.96 +23.02	543.45
DS-R1	16.45	32.20 +15.75	37.14 +20.69	38.77 +22.32	**40.00** +23.55	277.11
MCTS at depth 5 trained on ACE_{med} (Development Set)						
GPT-4o	12.68	28.04 +15.36	27.03 +14.35	**28.57** +15.89	27.31 +14.63	17.55
GPT-4.5	16.47	32.35 +15.88	37.58 +21.11	36.22 +19.75	**37.74** +21.27	32.65
o1	13.94	33.52 +19.58	37.78 +23.84	38.71 +24.77	**39.81** +25.87	575.36
DS-R1	16.45	37.97 +21.52	38.40 +21.95	40.58 +24.13	**44.26** +27.81	301.45
MCTS at depth 5 trained on ACE_{med} (Test Set)						
GPT-4o	13.33	26.94 +13.61	34.75 +21.42	30.59 +17.26	**35.79** +22.46	27.00
GPT-4.5	14.29	27.31 +13.02	35.29 +21.00	36.59 +22.30	**36.69** +22.40	35.56
o1	15.38	28.57 +13.19	36.73 +21.35	**38.71** +23.33	37.86 +22.48	526.43
DS-R1	16.00	31.93 +15.93	41.98 +25.98	42.06 +26.06	**43.75** +27.75	211.43

Table 1: Cross-model prompt optimization on Event Extraction: AC (F1) scores using different $\mathcal{M}_{task}$ (rows) and $\mathcal{M}_{opt}$ (columns). `#Output Tokens` is the average number of output tokens from the task model. Background shades indicate the choice of prompt optimizers: LRMs, LLMs, or no optimization. The best result is in **bold** per task model; the highest relative gain is **underlined**. LRMs serve as strong optimizers across all task models.

gle round of MCTS optimization (depth 1) changes this picture substantially.

On ACE_{low}, where only 15 training examples are available, both LRMs achieve the largest gains: DeepSeek-R1 improves from $16.45 \mapsto 24.66$ ($+8.21$) and o1 from $13.94 \mapsto 21.92$ ($+7.98$), compared to $+7.47$ for GPT-4o and $+5.84$ for GPT-4.5. On ACE_{med}, gains grow substantially across the board, with DeepSeek-R1 reaching 40.00 ($+23.55$) and o1 reaching 36.98 ($+23.04$). LLMs also benefit considerably (GPT-4.5: $+20.04$; GPT-4o: $+14.86$), though LRMs reach the highest post-optimization scores on ACE_{med}. After optimization, DeepSeek-R1 outperforms the best LLM by $+3.49$ AC on ACE_{med}, and these gains are statistically significant with non-overlapping 95% bootstrap confidence intervals. This gap widens further under full MCTS (RQ2).

> **Insight 1**: Prompt optimization benefits all models, but LRMs gain more, no matter whether small and medium-sized training data is present.

RQ2: How do LRMs perform under full-scale MCTS prompt optimization? To assess whether LRM advantages persist with deeper search, we run full MCTS (depth 5) on ACE_{med}.

Model	No Opt.	D1 (Dev)	D5 (Dev)	D5 (Test)
o3	19.63	25.61	48.95	45.00
o4-mini	20.25	28.53	50.11	46.51

Table 2: Results for newer LRMs under self-optimization on ACE_{med} (AC F1). Both models show large gains from MCTS-based prompt optimization, consistent with DeepSeek-R1 and o1.

Compared to depth 1, full MCTS yields further improvements, though the incremental gains are smaller than the initial jump from zero-shot to depth 1. LRMs, however, still exhibit stronger improvements: DeepSeek-R1 gains an additional $+4.26$ AC over its depth-1 best ($40.00 \mapsto 44.26$), and o1 improves by $+2.83$ ($36.98 \mapsto 39.81$). In contrast, GPT-4.5 and GPT-4o gain only $+1.23$ ($36.51 \mapsto 37.74$) and $+1.03$ ($27.54 \mapsto 28.57$), respectively. We evaluate each task model on the test set using the best prompt found during search on ACE_{med}. The pattern still holds: DeepSeek-R1 reaches 43.75 and o1 reaches 38.71, outperforming GPT-4.5 (36.69) and GPT-4o (35.79).

Given the rapid release of newer reasoning models, we additionally evaluate o3 and o4-mini on ACE_{med} under self-optimization (Table 2). Both models follow the same trend, showing large gains under prompt optimization.

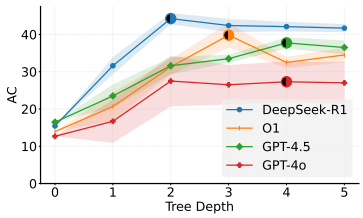
Insight 2: Full-scale MCTS optimization yields non-dramatic gains over single-step optimization, but LRMs benefit more.

RQ3: Do LRMs make better prompt optimizers? In the low-resource setting (ACE_{low}, Depth 1), DeepSeek-R1 is the best optimizer for every task model. Compared to the strongest LLM optimizer (GPT-4o), DeepSeek-R1 yields gains of $+1.97$ AC for GPT-4o ($18.18 \mapsto 20.15$), $+2.98$ for GPT-4.5 ($19.33 \mapsto 22.31$), $+2.96$ for o1 ($18.96 \mapsto 21.92$), and $+5.99$ when optimizing itself ($18.67 \mapsto 24.66$). Notably, GPT-4o outperforms GPT-4.5 as an optimizer across all task models, despite being a weaker task model itself.

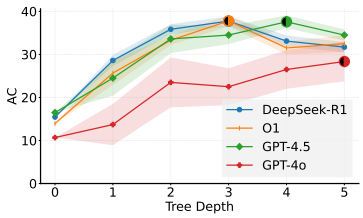
With more training data (ACE_{med}, Depth 1), the picture shifts. LRM optimizers remain strong, with both achieving over $+23$ AC gain when optimizing themselves. However, GPT-4.5 shows a substantial boost in optimizer effectiveness, consistently outperforming GPT-4o and narrowing the gap with LRMs, reaching 35.94 when optimizing itself and 36.67 when optimizing o1. Qualitatively, as shown in Table 3, DeepSeek-R1 enhances the optimized prompt $\mathcal{P}^*$ by adding precise extraction rules, such as removing articles ("a/an/the") and possessive pronouns (highlighted in blue), as well as critical exception cases for handling specific triggers (highlighted in pink). In contrast, o1 tends to generate more extraction rules, resulting in longer prompts. Both LRMs include specific examples to guide extraction. LLMs, by comparison, focus on task instructions and output formatting, typically generating shorter prompts with fewer examples. Among them, GPT-4.5 occasionally adds exception handling, though less consistently than LRMs. Additional examples of optimized task instructions and event guidelines are in Appendix E, with further analysis of prompt quality in Section 5.

Insight 3: LRMs serve as highly effective optimizers, especially in low-resource settings where DeepSeek-R1 consistently outperforms all others as a prompt optimizer.

RQ4: Can LRMs act as efficient and stable optimizers in prompt optimization? As shown in Fig. 4a, with DeepSeek-R1 as the optimizer, task models such as DeepSeek-R1 and GPT-4o converge faster than when GPT-4.5 serves as the optimizer (Fig. 4b), suggesting that DeepSeek-R1 generates higher-quality prompts. It also exhibits smaller performance variance for DeepSeek-R1 and GPT-4.5 as task models, indicating that DeepSeek-R1 produces effective prompts not only with higher quality but also more reliably. In contrast, with GPT-4.5 as the optimizer, convergence is slower: both LRMs peak at depth 3, while GPT-4.5 and GPT-4o converge at depths 4 and 5, respectively, with visibly less stable optimization. Most models also begin to plateau or slightly decline beyond their optimal depth (marked with half-filled markers), reinforcing the diminishing returns observed in RQ2.



(a) $\mathcal{M}_{opt}$ =DeepSeek-R1



(b) $\mathcal{M}_{opt}$ =GPT-4.5

Figure 4: Convergence analysis of prompt optimization across different task models with two optimizers: DeepSeek-R1 (left) and GPT-4.5 (right). Task models converge faster with minimal variance when optimized by LRMs.

Insight 4: DeepSeek-R1 (LRM) as an optimizer yields faster and more stable convergence than GPT-4.5 (LLM).

5 Further Analysis

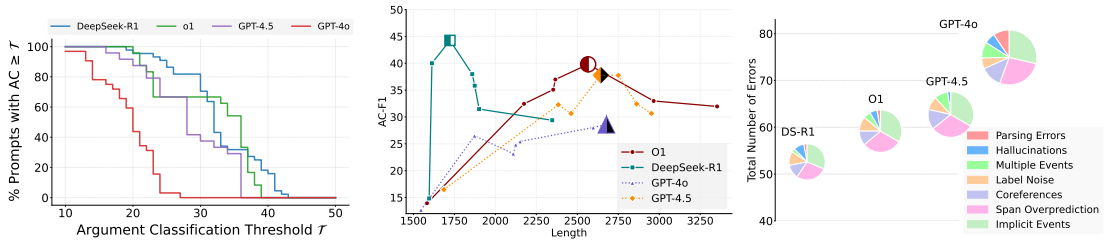
Prompt Quality Across Optimizers Beyond the qualitative analysis in RQ3, we examine the distribution of prompt quality using a survival plot (Fig. 5a) with DeepSeek-R1 as $\mathcal{M}_{task}$, showing the percentage of prompts that achieve at least a given AC threshold. Prompts optimized by DeepSeek-R1 exhibit the strongest survival curve, maintaining high-performance density even at stricter cutoffs ($\geq 35\%$ AC). GPT-4o's curve decays rapidly, indicating inconsistent prompt quality. o1 and GPT-4.5 fall in between, with o1 slightly outperforming GPT-4.5 at mid-range thresholds but trailing DeepSeek-R1 at higher cutoffs.

Examples of Task Instructions Optimized by Different Models	
NO OPTIMIZATION Best Scores TI - 39.29 TC - 33.93 AI - 16.47 AC - 16.47	# This is an event extraction task where the goal is to extract structured events from the text following structured event definitions in Python. **(...)** For each different event type, please output the extracted information from the text into a python list format **(...)** you should always output in a valid pydantic format: result = [EventName("mention" = "trigger", "arg1_key" = "arg1_span", ...), EventName("mention" = "trigger", "arg1_key" = "arg1_span", ...)]. **(...)**
GPT-4O Best Scores TI - 48.28 TC - 48.28 AI - 40.51 AC - 37.97	# This is an event extraction task where the goal is to extract structured events **(...)** # Task Instructions: 1. For each different event type, output the extracted information from the text **(...)** 2. Structure the output in a valid Pydantic format: 'result = [EventName("mention" = "trigger", **(...)**. 3. Adhere strictly to the described event descriptions **(...)**. 4. Address special cases:- Appeals: Consider involved parties from prior related events as "prosecutor". - Multiple roles may apply contextually; ensure complete information extraction. - Implicit indications: If mentions like "filed", "concluded", etc.,**(...)** use context to clarify them.**(...)**
GPT-4.5 Best Scores TI - 46.15 TC - 46.15 AI - 40.80 AC - 38.40	# This is an event extraction task for identifying and structuring events from text using Python-defined event classes. Each structured event consists of an event trigger word, an event type **(...)** ## Instructions: 1. Span Extraction: - Extract precise and concise spans for mentions and arguments, conveying the event or argument role clearly **(...)** - Accurately identify roles using contextual cues, effectively resolving ambiguities while prioritizing explicit spans. If roles are unmentioned, leave them empty. **(...)** 3. Output Format: Please follow the Python-format**(...)** 4. Clarifications and Exceptions:- Note explicitly when roles have exceptions based on role definitions. - Manage overlapping roles by following specific guidelines for span clarity and precision, **(...)**
DEEPSEEK-R1 Best Scores TI - 56.60 TC - 56.60 AI - 44.26 AC - 44.26	# Event Extraction Task: Extract structured events from text using Python class definitions.**(...)**: 1. Span Extraction:- Triggers: Minimal contiguous spans (verbs/nouns) directly expressing the event. Include both verbal and nominal forms **("death" = Die, "killings" = Die)**.**(...)** - Arguments: - Remove articles **("a/an/the")** and possessive pronouns EXCEPT when part of official names or temporal phrases **("The Hague", "the past year")** - Resolve pronouns AND POSSESSIVE NOUNS to named entities immediately using same-sentence antecedents **("airline's plan" → ["airline"])** - Strip role/location/age descriptors from arguments **("Philadelphia lawyers" → "lawyers")** **(...)** - Keep FULL spans for crimes/money including sources/amounts **("stereo worth $1,750 from family")** unless legal terms **(...)** 2. Special Handling:- Bankruptcy Triggers: **"went bust" → EndOrg(...)** - Crime Spans: Retain full contextual clauses **("If convicted of killings...")** without truncation - Temporal Phrases: Keep original spans with articles when part of phrase **("the early 90's")** 3. Output Rules: Always output in Python-format as **(...)** 4. Critical Exceptions:**-(...)**
O1 Best Scores TI - 66.67 TC - 66.67 AI - 44.93 AC - 40.58	# This is an event extraction task where the goal is to extract structured events from the text following structured event definitions in Python. **(...)** Keep argument references minimal by removing articles, possessives, or descriptive words unless they are crucial identifiers (e.g., **"the retailer" → "retailer"**, **"my uncle" → "uncle"**). # Important guidelines to address prior errors: # 1. For each event trigger, use the single most relevant word (e.g., **"bankruptcy"** rather than **"file for bankruptcy"**). # 2. For argument roles, also use minimal spans (e.g., **"soldier"** instead of **"a soldier," "woman"** instead of **"a woman"**).**(...)** # 4. For justice events (Sue, Appeal, Convict, SentenceAct, etc.): **(...)** # 5. For transfers of money, watch for direct or indirect references to donations, **(...)** # 6. Do not skip events implied by synonyms or indirect wording (e.g., **"shutting down"** → EndOrg, **(...)**. # 7. If there is more than one event in a single text, output each in a separate entry.**(...)**

Table 3: Example task instructions optimized by different optimizers when $\mathcal{M}_{task}$ = DeepSeek-R1, which yielded the best performance for each optimizer. LRMs tend to emphasize actionable extraction rules and exception handling, while paying minimal attention to the task instruction and output format. Additionally, they often include illustrative examples **(in bold)** to facilitate span extraction.

Prompt Length vs. Task Model Performance We analyze the relationship between prompt length and AC score across full MCTS search trees. For each model, we select its best-performing search trajectory (o1 as optimizer for GPT-4o, DeepSeek-R1 as optimizer for the other task models) and plot the full prompt lengths against AC scores in Fig. 5b. DeepSeek-R1 achieves its highest performance with the shortest prompt (∼1750 tokens), suggesting a preference for concise task instructions. In contrast, both LLMs and o1 rely on significantly longer prompts to reach their best accuracy.

Error Analysis We conduct a fine-grained analysis of all development examples where DeepSeek-R1 fails on prompts generated by different optimizers. As shown in Fig. 5c, LRM optimizers notably reduce event-related errors, particularly those involving multiple or implicit events. Argument-related issues, such as coreference errors and span overprediction, are also slightly reduced. In some cases, all models produce non-parsable outputs or hallucinated argument spans. The remaining errors are primarily attributed to label noise in the dataset. Examples for each error category are in Appendix C.



(a) (b) (c)

Figure 5: (a) Survival plot showing the % of prompts achieving at least a given AC score for DeepSeek-R1 across different optimizers. (b) Prompt length vs. AC score for each task model's best MCTS configuration on the dev set. (c) Error categorization for DeepSeek-R1 as the task model with various optimizers.

> **Insight 5**: LRM-optimized prompts are enriched with new extraction rules absent from the original task instruction, directly addressing frequent errors.

Batch Prompting in the Optimization Loop We integrate batch prompting (Cheng et al., 2023) into the MCTS loop by combining multiple queries into a single call during answer generation (Step 1 in Fig. 3), providing a 15-fold reduction in API calls at batch size 15. Comparing single-instance querying (BS-1) to batched variants, we find that every model achieves higher AC at some batch size than at BS-1 (e.g., GPT-4.5 improves from 22.1 $\mapsto$ 32.2 at BS-10, DS-R1 from 29.8 $\mapsto$ 30.7 at BS-15), though the optimal size is model-dependent with no universal heuristic. This accuracy improvement aligns with recent evidence that batch prompting constrains excessive reasoning in LRMs, reducing reasoning tokens extensively while preserving task accuracy (Srivastava et al., 2025a). We select BS-15 because it maximizes cost reduction while yielding the best AC for DeepSeek-R1, our strongest task model. Full batch size analysis is in Appendix A.2.

Generalization Beyond Event Extraction As shown in Table 4, prompt optimization consistently improves all models across all three tasks. On Geometric Shapes, o1 and DeepSeek-R1 reach test accuracies of 77.80 and 78.40, outperforming GPT-4.5 (74.20) and GPT-4o (67.50). While GPT-4o achieves the largest relative gain (+14.10), LRMs still achieve higher absolute performance. On NCBI, LRMs show strong gains: o1 and DeepSeek-R1 improve by +17.02 and +15.76 F1, reaching 70.15 and 69.96, well above LLM performance. Gridworld Planning is particularly noteworthy as it involves no schema or predefined labels and requires multi-step action sequences with multiple valid solutions. Here, DeepSeek-R1 improves from 54.00 $\mapsto$ 88.00, substantially outperforming GPT-4o (41.00 $\mapsto$ 71.00).

Model	No Opt.	D1 (Dev)	D5 (Dev)	D5 (Test)
(a) Geometric Shapes (Accuracy)				
GPT-4o	53.40	61.20 +7.8	68.67 +15.3	67.50 +14.1
GPT-4.5	69.96	72.90 +2.9	75.33 +5.4	74.20 +4.2
o1	**70.07**	73.50 +3.4	78.00 +7.9	77.80 +7.7
DS-R1	69.67	**73.80** +4.1	**78.67** +9.0	**78.40** +8.7
(b) NCBI Disease NER (Micro-F1)				
GPT-4o	43.75	49.50 +5.8	54.37 +10.6	52.63 +8.9
GPT-4.5	**56.25**	58.67 +2.4	65.56 +9.3	64.56 +8.3
o1	53.13	**66.46** +13.3	**71.46** +18.3	**70.15** +17.0
DS-R1	54.20	66.00 +11.8	71.40 +17.2	69.96 +15.8
(c) Gridworld Planning (Exact-Match Acc.)				
GPT-4o	41.00	48.00 +7.0	73.00 +32.0	71.00 +30.0
DS-R1	**54.00**	**67.00** +13.0	**89.00** +35.0	**88.00** +34.0

Table 4: Generalization results on symbolic reasoning, biomedical NER, and open-ended planning. Prompt optimization consistently improves all models, with LRMs achieving the highest post-optimization performance. For Gridworld, we evaluate GPT-4o and DeepSeek-R1 due to API cost and compute constraints.

6 Conclusion

We present the first systematic study of prompt optimization for LRMs, evaluating their roles as both task models and optimizers in a unified MCTS-based framework. On end-to-end EE, all models benefit substantially from optimization, with LRMs achieving the highest post-optimization scores. LRMs also serve as stronger optimizers, producing shorter, more rule-oriented prompts that converge faster with lower variance. These findings generalize across Geometric Shapes, NCBI NER, and Gridworld Planning, and hold across six models, including the latest o3 and o4-mini, establishing that prompt optimization remains both essential and effective in the era of reasoning models.

Limitations

Our study has several limitations. First, due to cost and compute constraints, we evaluate o3 and o4-mini only in self-optimization mode on a single task (Event Extraction), and do not run the full cross-model optimizer grid for these newer models. Second, DeepSeek-R1 is deployed locally with 2.5-bit quantization, which may underestimate its full capability compared to full-precision or API-based inference. Third, while we evaluate across four diverse tasks, our deepest analysis (full MCTS depth-5, convergence, error categorization) is conducted on Event Extraction; the generalization tasks use shallower experimental configurations. Fourth, our MCTS-based framework is one of several prompt optimization strategies, and future work should investigate whether our findings hold under alternative optimization paradigms such as TextGrad (Yuksekgonul et al., 2024) or Bayesian approaches like MIPROv2 (Opsahl-Ong et al., 2024). Finally, our experiments focus on English-language tasks and closed/open-weight models accessible at the time of writing; extending these findings to multilingual settings and emerging LRMs remains an open direction.

Ethics Statement

This work studies automated prompt optimization for large language and reasoning models on publicly available benchmarks. We use the ACE05 dataset under its standard LDC license and do not introduce new data collection involving human subjects. Our generalization benchmarks (Geometric Shapes, NCBI Disease NER, Gridworld Planning) are likewise publicly released datasets. All model outputs are evaluated automatically using established metrics; no human evaluation or annotation was conducted for this study. We note that prompt optimization could, in principle, be used to elicit harmful or biased outputs from language models; however, our framework is designed and evaluated exclusively for improving task accuracy on well-defined extraction and reasoning benchmarks. DeepSeek-R1 is deployed locally on our own infrastructure, and all API-based models (GPT-4o, GPT-4.5, o1, o3, o4-mini) are accessed through their official APIs in accordance with their terms of use.

References

Mohamed Aghzal, Erion Plaku, and Ziyu Yao. 2024. Can large language models be good path planners? a benchmark and investigation on spatial-temporal reasoning. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.

Zhoujun Cheng, Jungo Kasai, and Tao Yu. 2023. Batch prompting: Efficient inference with large language model APIs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 792–810, Singapore. Association for Computational Linguistics.

Michael Han Daniel Han and Unsloth team. 2023. Unsloth.

George Doddington, Alexis Mitchell, Mark Przybocki, Lance Ramshaw, Stephanie Strassel, and Ralph Weischedel. 2004. The automatic content extraction (ACE) program – tasks, data, and evaluation. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC'04)*, Lisbon, Portugal. European Language Resources Association (ELRA).

Rezarta Islamaj Doğan, Robert Leaman, and Zhiyong Lu. 2014. Ncbi disease corpus: a resource for disease name recognition and concept normalization. *Journal of biomedical informatics*, 47:1–10.

Jun Gao, Huan Zhao, Wei Wang, Changlong Yu, and Ruifeng Xu. 2024. Eventrl: Enhancing event extraction with outcome supervision for large language models.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. 2024. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *The Twelfth International Conference on Learning Representations*.

Kuan-Hao Huang, I-Hung Hsu, Tanmay Parekh, Zhiyu Xie, Zixuan Zhang, Prem Natarajan, Kai-Wei Chang, Nanyun Peng, and Heng Ji. 2024. Textee: Benchmark, reevaluation, reflections, and future challenges in event extraction. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 12804–12825.

Omar Khattab, Arnav Singhvi, Hanna Maheshwari, Keshav Zhang, Siddharth Santhanam, Estevam Hruschka, Christopher Potts, and Matei Zaharia. 2026. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*.

Peng Li, Tianxiang Sun, Qiong Tang, Hang Yan, Yuanbin Wu, Xuanjing Huang, and Xipeng Qiu. 2023. CodeIE: Large code generation models are better few-shot information extractors. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15339–15353, Toronto, Canada. Association for Computational Linguistics.

Wenwu Li, Xiangfeng Wang, Wenhao Li, and Bo Jin. 2025. A survey of automatic prompt engineering: An optimization perspective.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, Online. Association for Computational Linguistics.

Zixuan Li, Yutao Zeng, Yuxin Zuo, Weicheng Ren, Wenxuan Liu, Miao Su, Yucan Guo, Yantao Liu, Lixiang Lixiang, Zhilei Hu, Long Bai, Wei Li, Yidan Liu, Pan Yang, Xiaolong Jin, Jiafeng Guo, and Xueqi Cheng. 2024. KnowCoder: Coding structured knowledge into LLMs for universal information extraction. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8758–8779, Bangkok, Thailand. Association for Computational Linguistics.

Agustin Mantaras. 2025. Prompt engineering for openai's o1 and o3-mini reasoning models. Microsoft Tech Community Blog.

Hector D. Menendez, Gema Bello-Orgaz, and Cristian Ramírez Atencia. 2025. Deepstableyolo: Deepseek-driven prompt engineering and search-based optimization for AI image generation. In *XVI Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados*.

OpenAI. 2025. Reasoning best practices. OpenAI Platform Documentation.

Krista Opsahl-Ong, Michael J. Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. 2024. Optimizing instructions and demonstrations for multi-stage language model programs. *arXiv preprint arXiv:2406.11695*.

Tanmay Parekh, Kartik Mehta, Ninareh Mehrabi, Kai-Wei Chang, and Nanyun Peng. 2025. DiCoRe: Enhancing zero-shot event detection via divergent-convergent LLM reasoning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 20560–20582, Suzhou, China. Association for Computational Linguistics.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7957–7968, Singapore. Association for Computational Linguistics.

Qwen Team. 2025. QwQ-32B: Embracing the power of reinforcement learning. `https://qwenlm.github.io/blog/qwq-32b/`. Blog post.

Oscar Sainz, Iker García-Ferrero, Rodrigo Agerri, Oier Lopez de Lacalle, German Rigau, and Eneko Agirre. 2024. GoLLIE: Annotation guidelines improve zero-shot information-extraction. In *The Twelfth International Conference on Learning Representations*.

Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. 2020. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4222–4235, Online. Association for Computational Linguistics.

Saurabh Srivastava, Janit Bidhan, Hao Yan, Abhishek Dey, Tanu Kansal, Paras Kath, Sina Mansouri, Mohit Marvania, Vamsi Shankar Simhadri, and Gaurav Singh. 2025a. Reasoning under constraint: How batch prompting suppresses overthinking in reasoning models. *arXiv preprint arXiv:2511.04108*.

Saurabh Srivastava, Chengyue Huang, Weiguo Fan, and Ziyu Yao. 2024. Instances need more care: Rewriting prompts for instances with LLMs in the loop yields better zero-shot performance. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 6211–6232, Bangkok, Thailand. Association for Computational Linguistics.

Saurabh Srivastava, Sweta Pati, and Ziyu Yao. 2025b. Instruction-tuning llms for event extraction with annotation guidelines.

Saurabh Srivastava, Gaurav Singh, Shou Matsumoto, Ali Raz, Paulo Costa, Joshua Poore, and Ziyu Yao. 2023. MailEx: Email event and argument extraction. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 12964–12987, Singapore. Association for Computational Linguistics.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, and 1 others. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*.

Together AI. 2025. Prompting deepseek-r1. Together AI Documentation.

Guoqing Wang, Zeyu Sun, Zhihao Gong, Sixiang Ye, Yizhou Chen, Yifan Zhao, Qingyuan Liang, and Dan Hao. 2024a. Do advanced language models eliminate the need for prompt engineering in software engineering?

Xingyao Wang, Sha Li, and Heng Ji. 2023. Code4struct: Code generation for few-shot event structure prediction. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3640–3663.

Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric Xing, and Zhiting Hu. 2024b. Promptagent: Strategic planning with language models enables expert-level prompt optimization. In *The Twelfth International Conference on Learning Representations*.

Kangda Wei, Aayush Gautam, and Ruihong Huang. 2024. Are LLMs good annotators for discourse-level event relation extraction? In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 1–19, Miami, Florida, USA. Association for Computational Linguistics.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2024. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. Textgrad: Automatic "differentiation" via text. *Nature*.

Tianyang Zhong, Zhengliang Liu, Yi Pan, Yutong Zhang, Yifan Zhou, Shizhe Liang, Zihao Wu, Yanjun Lyu, Peng Shu, Xiaowei Yu, and 1 others. 2024. Evaluation of openai o1: Opportunities and challenges of agi.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*.

A Additional Experimental Details

A.1 MCTS Algorithm

To effectively optimize prompts for task-specific performance, we adopt a Monte Carlo Tree Search (MCTS) framework that iteratively explores and refines prompts based on model feedback and reward signals. The proposed algorithm, outlined in Algorithm 1, combines structured exploration with guided optimization by leveraging a task model, a feedback-generating optimizer, and a reward function. At each iteration, the algorithm performs selection, expansion, simulation, and back-propagation steps, progressively improving the prompt to maximize task performance across sampled batches.

Algorithm 1 Algorithm for MCTS-based Prompt Optimization

Inputs:
Initial prompt $s_0 = \mathcal{P}_0$, task model $\mathcal{M}_{task}$, optimizer $\mathcal{M}_{opt}$, reward function $\mathcal{R}$, batch size k, depth limit L, iterations τ, exploration weight c
Initialize:
State-action mapping $A : \mathcal{S} \mapsto \mathcal{F}$, children mapping ch : $\mathcal{S} \times \mathcal{F} \mapsto \mathcal{S}$, rewards $r : \mathcal{S} \times \mathcal{F} \mapsto \mathbb{R}$,
Q-values $Q : \mathcal{S} \times \mathcal{F} \mapsto \mathbb{R}$, visit count $\mathcal{N} : \mathcal{S} \mapsto \mathbb{N}$
for $n \leftarrow 0, \ldots, \tau - 1$ **do**
  Sample batch (Q_{batch}, A_{batch}) from training data
  for $t \leftarrow 0, \ldots, L - 1$ **do**
    if $A(s_t)$ is not empty **then** ▷ selection
      $f_t \leftarrow \arg\max_{f \in A(s_t)} \left(Q(s_t, f) + c \cdot \sqrt{\frac{\ln \mathcal{N}(s_t)}{\mathcal{N}(\text{ch}(s_t, f))}} \right)$
      $s_{t+1} \leftarrow \text{ch}(s_t, f_t), r_t \leftarrow r(s_t, f_t), \mathcal{N}(s_t) \leftarrow \mathcal{N}(s_t) + 1$
    else ▷ expansion and simulation
      (Step 1) Answer Gen: $\hat{Q}_{batch} \sim \mathcal{M}_{task}(Q_{batch}, s_t)$
      (Step 2) Error Extract: Identify errors using interpreter on $\hat{A}_{batch}$
      (Step 3) Feedback Gen: $f_t \sim \mathcal{M}_{opt}(\text{feedback}|s_t, \text{errors})$
      (Step 4) Prompt Update: $s_{t+1} \sim \mathcal{M}_{opt}(s|s_t, f_t)$
      Update $A(s_t) \leftarrow \{f_t\}, \text{ch}(s_t, f_t) \leftarrow s_{t+1}, r(s_t, f_t) \leftarrow \mathcal{R}(\hat{A}_{batch}, A_{batch})$
      $r_t \leftarrow r(s_t, f_t), \mathcal{N}(s_t) \leftarrow \mathcal{N}(s_t) + 1$
    end if
    if s_{t+1} is an early-stopping state **then**
      break
    end if
  end for
  $T \leftarrow$ number of steps
  for $t \leftarrow T - 1, \ldots, 0$ **do** ▷ back-propagation
    Update $Q(s_t, f_t)$ with rollout rewards $\{r_t, \ldots, r_L\}$
  end for
end for

A.2 Batch Prompting Details

Since querying LLMs individually for each input incurs substantial computational costs, a naïve approach that treats each input separately is inefficient. To mitigate this, we employ **batch prompting** (Cheng et al., 2023), which enables the combination of multiple queries into a single structured prompt. Given a batch of inputs $\{Q_1, Q_2, \ldots, Q_n\}$ that share the same task instruction $\mathcal{P}_{\mathcal{I}}$, batch prompting constructs a concatenated input string in the form $[\mathcal{P}_0 \| Q_1 \| Q_2 \| \ldots \| Q_n]$. Each query is uniquely labeled (e.g., "text1") to maintain order and structure. The model processes this batch and generates a structured response in the form $[A_1 \| A_2 \| \ldots \| A_n]$, where each A_i corresponds to the output for Q_i. These responses are parsed to extract individual predictions while preserving alignment.

Batch Size Selection. Growing a full MCTS tree for prompt optimization can be computationally expensive (Wang et al., 2024b). We experimented with different batch sizes for constructing Q_{batch} by varying the number of queries Q_i and corresponding outputs A_i. However, we found that determining an optimal batch size is highly model-dependent and lacks a universal heuristic (Figure 6). We set the batch size to 15, as it provides a straightforward 15-fold reduction in API calls while maintaining response quality.

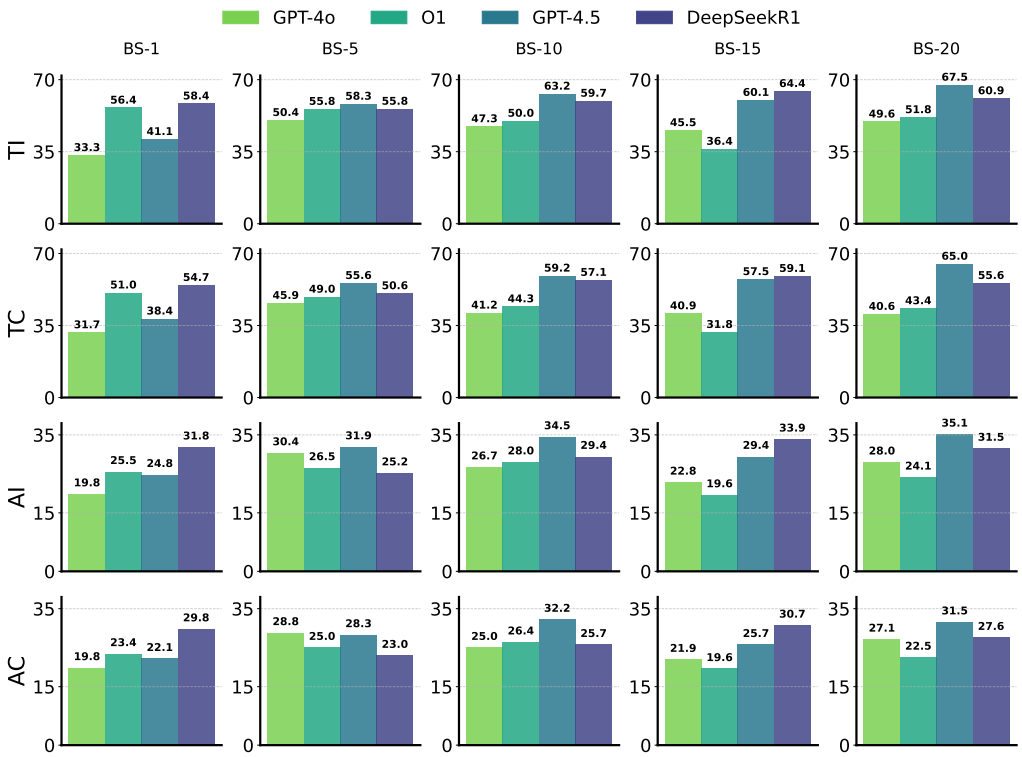


Figure 6: Batch-wise performance across different batch sizes and models. We select batch size 15 as a practical trade-off between cost reduction and accuracy.

A.3 Data Split and Event Type Distribution

We utilized two subsets of ACE05: ACE_{low} and ACE_{med}, described in Section 4.1. Table 5 presents the distribution of selected event types (ETs) across ACE_{low}, ACE_{med}, and the development set. These subsets span both frequent ETs (e.g., *SentenceAct*, *Die*) and rare ones (e.g., *PhoneWrite*, *DeclareBankruptcy*). The *None* class includes instances without annotated events, preserving realistic class distribution.

A.4 Code Prompt Example

Figure 7 shows a representative code-based prompt used in our EE experiments. The prompt consists of two components: (1) a **task instruction** that specifies the extraction objective, output format, and general extraction constraints, and (2) an **event schema** that encodes event types and their argument roles as Python class definitions, each paired with natural-language guidelines describing the event semantics and argument constraints. During prompt optimization, the MCTS framework refines both the task instruction and the event guidelines, while the Python class structure (event type names, argument role names) remains fixed to preserve schema consistency. The optimizer is free to add, remove, or rewrite extraction rules, exception cases, and illustrative examples within the guideline text.

A.5 Hyperparameters and MCTS Configuration

Following Wang et al. (2024b), we fix the depth limit $L = 5$ and MCTS iterations $\tau = 12$. The exploration weight is $c = 2.5$. The batch size k for each rollout is 15. We use greedy decoding for $\mathcal{M}_{task}$ and temperature sampling with $T = 0.7$ for $\mathcal{M}_{opt}$ to promote diverse feedback. Early stopping is triggered if a prompt leads to zero errors across two consecutive rollouts.

A.6 Generalization Task Details

Prompt Format. For Event Extraction, we follow established prior approaches and use a code-based prompting format with Python class definitions and annotation guidelines (Srivastava et al., 2025b). For Geometric Shapes, NCBI Disease NER, and Gridworld Planning, we retain the original natural-language

	Train ACE_{low}	Train ACE_{med}	Dev
TransferMoney	3	13	29
Meet	2	15	13
PhoneWrite	1	11	1
SentenceAct	6	25	4
Appeal	2	16	4
Convict	5	11	5
Sue	3	13	8
EndOrg	1	11	1
Die	2	26	15
DeclareBankruptcy	1	11	1
None	5	20	30

Table 5: Data distribution for selected ETs.

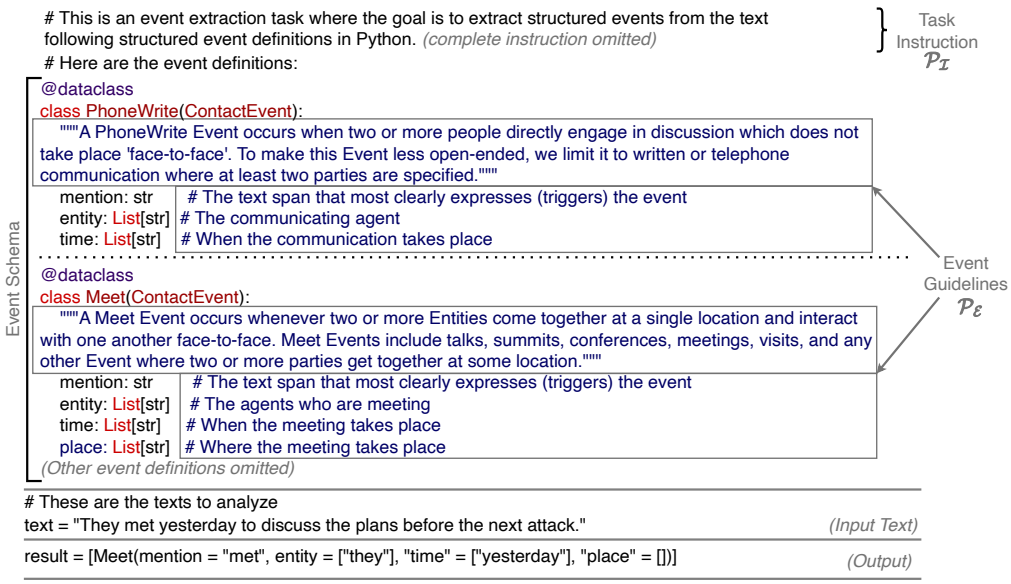


Figure 7: An example prompt for end-to-end Event Extraction (EE) used in our experiments, consisting of a task instruction and an event schema. The event schema contains information about the labels that are represented as Python classes and event guidelines defining both the event classes and the arguments. In prompt optimization, we refine both the task instruction and event guidelines (shown for two events; others omitted due to space limits) to generate more effective prompts for the task model.

prompts provided by the dataset authors. Despite these differences in surface form, the qualitative trends remain identical: LRMs benefit more from optimization and serve as stronger optimizers.

Gridworld Planning Setup. Following Aghzal et al. (2024), we use a subset of 100 randomly sampled instances from the single-goal PNNL test set (unseen environments) and 1,000 examples from the training set. Both datasets consist of 6×6 environments with 1–5 obstacles. This benchmark is substantially more open-ended than the other tasks because (i) it has no schema or predefined label inventory, (ii) it requires multi-step action sequences rather than classifying spans or triggers, and (iii) it admits multiple valid solution paths. We evaluate using task completion accuracy.

o3 and o4 Setup. Due to API cost constraints, we evaluate o3 and o4 only in self-optimization mode ($\mathcal{M}_{opt} = \mathcal{M}_{task}$) on ACE$_{med}$. All other hyperparameters and MCTS configurations remain identical to the main experiments.

B Statistical Robustness

B.1 Bootstrap Confidence Intervals

To confirm the robustness of our main findings, we compute non-parametric bootstrap 95% confidence intervals (**?**) for our core configuration (DeepSeek-R1 as both optimizer and task model on ACE$_{med}$). We resample the test set with replacement (1,000 resamples) and recompute AC F1 on each sample using the stored predictions.

Setting	AC (Test)	95% CI	Δ vs. No Opt
No Opt	16.00	[13.50, 17.60]	–
Depth-1	35.20	[32.53, 39.62]	+19.20
Depth-5	43.75	[40.80, 47.60]	+27.75

Table 6: Bootstrap 95% confidence intervals for DeepSeek-R1 (self-optimization) on ACE$_{med}$ test set. All intervals are non-overlapping, confirming that the gains from prompt optimization are statistically significant.

The confidence intervals are narrow and non-overlapping. The gain from No Opt to Depth-1 (+19.20) and No Opt to Depth-5 (+27.75) both far exceed the uncertainty ranges (<4 points). Even the Depth-1 to Depth-5 improvement (+8.55) exceeds the combined CI widths, indicating a meaningful and robust improvement.

B.2 Reward Function Ablation

As discussed in Section 4.1, our MCTS reward function uses the average of F1 scores across all four EE subtasks (TI, TC, AI, AC). To justify this choice over optimizing AC directly, we conduct an ablation using DeepSeek-R1 as both optimizer and task model on ACE$_{med}$.

EE is an inherently pipeline-coupled task: TI → TC → AI → AC. Errors in earlier subtasks propagate into AC; optimizing AC alone encourages the optimizer to overfit the final subtask while neglecting foundational ones. As shown in Table 7, averaged F1 consistently yields the best AC scores at both depths. The weighted variant (.3/.3/.2/.2) gives slightly higher emphasis to upstream subtasks but does not outperform equal weighting on AC.

C Full Event Extraction Results

We present complete results across all four EE metrics (TI, TC, AI, AC) for every model×optimizer combination. Tables 8–11 cover ACE$_{low}$ depth-1, ACE$_{med}$ depth-1, ACE$_{med}$ depth-5 (dev), and ACE$_{med}$ depth-5 (test), respectively.

C.1 How Do Optimizers Follow (or Ignore) Feedback?

Optimizers exhibit different behaviors in how they apply feedback. DeepSeek-R1 refines only the event definitions explicitly mentioned in the feedback, leaving the remaining untouched. An example is shown

Reward Type	TI	TC	AI	AC
Depth-1				
Average (ours)	37.50	33.93	25.00	**24.66**
AC-only	33.26	31.09	18.78	17.37
Weighted (.3/.3/.2/.2)	41.89	37.57	23.08	21.72
Depth-5				
Average (ours)	56.60	56.60	44.26	**44.26**
AC-only	44.44	44.44	30.11	29.03
Weighted (.3/.3/.2/.2)	59.26	59.26	37.58	36.24

Table 7: Reward function ablation (DeepSeek-R1 self-optimization on ACE_{med}). AC-only optimization performs worst because the model deprioritizes upstream subtasks (TI/TC), causing cascading drops. Averaged F1 yields the strongest and most stable AC results.

in Figure 8, where DeepSeek-R1 reasons that the incorrect argument extraction for the `Attack` event likely stems from limitations of $\mathcal{M}_{task}$ rather than the guideline itself, and consequently refuses to modify it.

> *(truncated)* The model's prediction didn't include this, possibly because the coref wasn't resolved. The current guideline says pronouns require coref resolution before extraction. So if "his" is unresolved, it's omitted. But in the correct answer, "his" is resolved to Arafat. So the model might not have access to coref resolution, leading to omission. However, this is a model limitation, but the guideline says pronouns must be coref resolved. So maybe the guideline is clear, but the model isn't performing coref properly. Hence, I will skip the Attack event. *(truncated)*

Figure 8: Example reasoning when DeepSeek-R1 refuses to edit the Attack event guideline.

To quantify this behavior, we measure the average number of edited guidelines and their average token length across all optimizers (Figure 9). DeepSeek-R1 edits the fewest event types (6.7 on average) and produces the shortest guidelines ($\sim$1.5k tokens), reflecting a feedback-sensitive and token-efficient strategy. In contrast, o1 and GPT-4.5 modify nearly all ten guidelines (9.8 and 8.5 on average), regardless of feedback specificity, resulting in much longer outputs (2.9k and 2k tokens). GPT-4o also appears restrained (7.6 edits on average), but qualitative analysis suggests this is due to feedback overflow rather than selectivity.

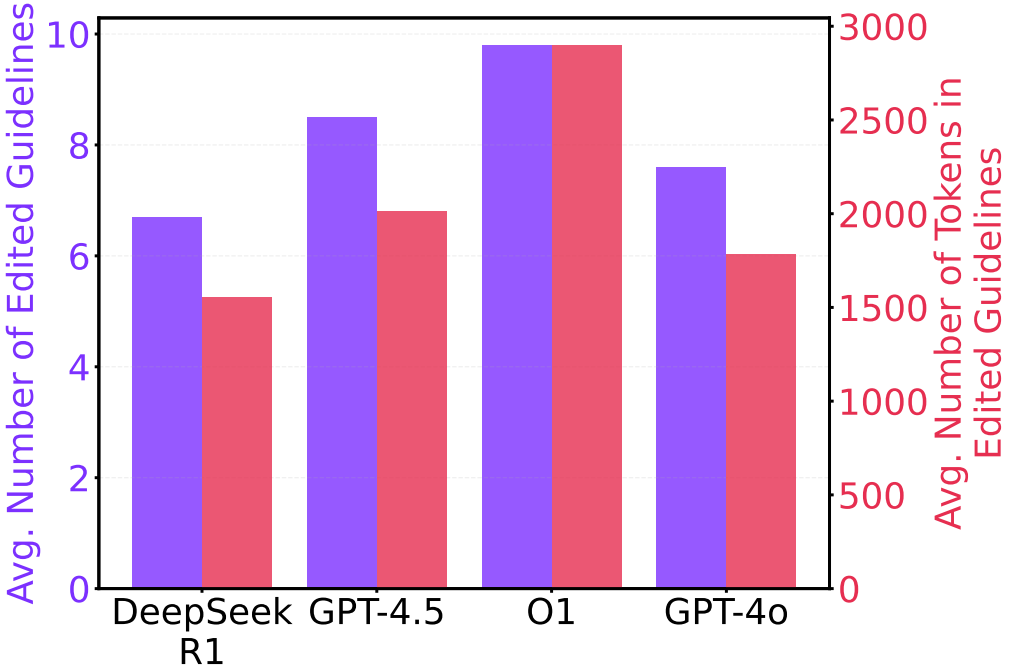


Figure 9: Average number of guidelines edited by each optimizer and average token count of edited guidelines ($\mathcal{M}_{task}$=DeepSeek-R1).

C.2 Full Results Tables

Models	DeepSeek-R1 (Optimizer)				o1 (Optimizer)				GPT-4.5 (Optimizer)				GPT-4o (Optimizer)			
	TI	TC	AI	AC	TI	TC	AI	AC	TI	TC	AI	AC	TI	TC	AI	AC
DeepSeek-R1	37.5	33.93	25.57	24.66	27.72	25.74	18.67	18.67	36.89	34.95	22.91	22.91	32.78	32.78	21.83	21.83
o1	31.54	31.54	21.92	21.92	29.33	29.33	18.96	18.96	31.91	31.91	18.57	18.57	29.24	29.24	21.74	20.29
GPT-4.5	36.04	34.23	23.14	22.31	34.78	33.04	20.07	19.33	31.37	31.37	19.32	19.32	30.29	30.29	20.97	20.19
GPT-4o	35.29	35.29	22.07	20.15	28.28	28.28	18.18	18.18	30.61	30.61	16.67	16.67	31.67	31.67	19.57	18.83

Table 8: Complete results of training on ACE$_{low}$ with MCTS depth 1 and tested on the dev set.

Models	DeepSeek-R1 (Optimizer)				o1 (Optimizer)				GPT-4.5 (Optimizer)				GPT-4o (Optimizer)			
	TI	TC	AI	AC	TI	TC	AI	AC	TI	TC	AI	AC	TI	TC	AI	AC
DeepSeek-R1	63.16	63.16	40.00	40.00	65.45	65.45	32.2	32.2	56.25	56.25	37.14	37.14	62.7	62.7	40.06	38.77
o1	78.95	78.95	39.13	36.96	54.78	54.78	33.96	30.19	59.26	59.26	36.67	36.67	57.14	57.14	36.98	36.98
GPT-4.5	64.71	64.71	35.42	35.42	46.15	46.15	29.63	29.63	63.57	63.57	35.94	35.94	59.21	59.21	38.1	36.51
GPT-4o	30.00	30.00	25.88	25.1	28.57	28.57	22.32	22.32	34.55	34.55	27.54	27.54	29.38	29.38	26.99	26.3

Table 9: Complete results of training on ACE_{med} with MCTS depth 1 and tested on the dev set.

Models	DeepSeek-R1 (Optimizer)				o1 (Optimizer)				GPT-4.5 (Optimizer)				GPT-4o (Optimizer)			
	TI	TC	AI	AC	TI	TC	AI	AC	TI	TC	AI	AC	TI	TC	AI	AC
DeepSeek-R1	56.6	56.6	44.26	44.26	66.67	66.67	44.93	40.58	46.15	46.15	40.8	38.4	48.28	48.28	40.51	37.97
o1	48.08	48.08	40.74	39.81	42.86	42.86	38.71	38.71	84.68	84.68	41.48	37.78	48.28	48.28	34.64	33.52
GPT-4.5	45.68	45.68	38.36	37.74	51.24	51.24	36.22	36.22	59.26	59.26	36.24	37.58	41.18	41.18	32.35	32.35
GPT-4o	49.09	49.09	28.11	27.31	61.11	61.11	28.57	28.57	52.00	52.00	27.03	27.03	61.54	61.54	29.91	28.04

Table 10: Complete results of training on ACE_{med} with MCTS depth 5 and tested on the dev set.

Models	DeepSeek-R1 (Optimizer)				o1 (Optimizer)				GPT-4.5 (Optimizer)				GPT-4o (Optimizer)			
	TI	TC	AI	AC	TI	TC	AI	AC	TI	TC	AI	AC	TI	TC	AI	AC
DeepSeek-R1	69.23	67.69	44.33	43.75	54.12	54.12	42.06	42.06	52.8	52.8	41.98	41.98	47.27	47.27	33.61	31.93
o1	68.28	67.76	38.44	37.86	67.86	67.86	38.71	38.71	58.29	58.29	36.73	36.73	41.11	41.11	28.57	28.57
GPT-4.5	68.31	68.31	38.44	36.69	64.71	64.71	39.02	36.59	56.45	56.45	35.29	35.29	49.09	49.09	28.11	27.31
GPT-4o	59.44	59.44	36.99	35.71	64.52	64.52	30.59	30.59	56.57	56.57	34.75	34.75	48.19	48.19	26.94	26.94

Table 11: Complete results of training on ACE_{med} with MCTS depth 5 and tested on the test set.

C.3 Error Categories and Examples

To complement the aggregate error analysis in Section 5, we manually inspected all development-set instances where DeepSeek-R1 produced incorrect predictions under prompts from each optimizer. We identified seven recurring error categories: **Parsing Errors** (non-parsable model output), **Hallucinations** (predicted events or arguments unsupported by the input), **Multiple Events** (failure to detect more than one event in a sentence), **Label Noise** (annotation inconsistencies in the gold data), **Coreferences** (unresolved pronominal or role-based references), **Span Overprediction** (argument spans that include unnecessary modifiers), and **Implicit Events** (events triggered by non-verbal cues such as adjectives or nouns). Table 12 provides a description and representative example for each category. As discussed in the main text, LRM-optimized prompts notably reduce event-related errors (multiple and implicit events) and argument-level issues (coreference and span overprediction), while parsing errors, hallucinations, and label noise persist across all optimizers.

Error Category	
Parsing Errors	**Description:** Parsing errors occur when the model's output is not in the expected format (e.g., JSON or structured list), often due to extra reasoning or verbose responses in batch prompts. These make the output unusable for evaluation pipelines.
	Example: Prompts that return extra text or commentary instead of a valid Python structure, causing non-parsable output.
Hallucinations	**Description:** Hallucinations occur when the model generates arguments or events that are not supported by the input. This usually happens due to biases learned during training or lexical overlaps with known labels.
	Example: **Text:** "Different parts of the strip saw conflicts today." → Model incorrectly predicts a 'Conflict' event based solely on the word "conflict".
Multiple Events	**Description:** Multiple event errors happen when the model detects only a single event in a sentence that contains multiple, usually defaulting to the most salient or final event.
	Example: **Text:** "...went home and his father-in-law killed him." → Model only predicts the 'Die' event, ignoring the 'Transport' event.
Label Noise	**Description:** Label noise refers to inconsistencies or ambiguities in the dataset annotations, such as differing treatment of coreferences or unclear event boundaries, which confuse both training and evaluation.
	Example: **Text:** "Our president has repeatedly... relied on a man... Hussein Kamel... leader of the Iraq arms program who defected..." → Label uses 'person=["leader"]'; model uses 'person=["Hussein Kamel"]'.
Coreferences	**Description:** Coreference errors arise when the model fails to resolve references like pronouns or role-based descriptors to their actual entities, leading to incorrect or incomplete argument spans.
	Example: **Text:** "...Hussein Kamel, leader of the Iraq arms program who defected..." → Label uses "leader"; model uses "Hussein Kamel", highlighting coreference resolution challenges.
Span Overprediction	**Description:** Span overprediction occurs when the model predicts more detailed argument spans than necessary, often including modifiers or descriptors not required by the task's minimal span rules.
	Example: **Text:** "Orders went out today to deploy 17,000 U.S. Army soldiers in the Persian Gulf region." → Label: "soldiers"; Prediction: "17,000 U.S. Army soldiers" – includes extra modifiers.
Implicit Events	**Description:** Implicit events are those not directly triggered by verbs but inferred through adjectives, nouns, or other context (e.g., "former"). These are often missed by models unless explicitly instructed.
	Example: **Text:** "...with former Congressman Tom Andrews..." → Trigger "former" implies 'EndPosition', but is often missed by models lacking rules for implicit event detection.

Table 12: Description of error categories with examples.

D Meta-Prompts

Our MCTS-based prompt optimization framework relies on two meta-prompts that govern how the optimizer model $\mathcal{M}_{opt}$ interacts with error signals to refine task prompts. These meta-prompts correspond to Steps 3 and 4 in Fig. 3.

Feedback Collection Prompt (m_{fb}). This meta-prompt (shown below) is used in Step 3 to elicit structured diagnostic feedback from $\mathcal{M}_{opt}$. Given the current prompt $\mathcal{P}_t$, the event guidelines $\mathcal{P}_{\mathcal{E}}$, and a set of incorrectly predicted examples from the current batch, the optimizer is asked to: (1) classify each error by type (e.g., incorrect span, missing role, format violation), (2) perform root cause analysis to determine whether the error stems from ambiguous guidelines, overlapping role definitions, or insufficient extraction rules, (3) propose example-specific fixes with "what to do" and "what not to do" instructions, and (4) identify recurring patterns across all errors. The structured output enables targeted, actionable prompt edits rather than generic rewrites.

Prompt Optimization Prompt (m_{opt}). This meta-prompt (shown below) is used in Step 4 to produce the refined prompt $\mathcal{P}_{t+1}$. It receives the current prompt, event guidelines, the error examples, the feedback f_t from Step 3, and the full trajectory of prior prompts to discourage redundant edits. The optimizer is instructed to: (1) refine the task instruction $\mathcal{P}_{\mathcal{I}}$ to address the identified issues, (2) update the natural-language guidelines within event class docstrings while preserving the Python class structure (event types and argument roles), and (3) maintain backward compatibility so that previously correct predictions remain valid. The output is structured with explicit delimiters (`<START>`/`<END>` for task instructions, `<CLASS_START>`/`<CLASS_END>` for event guidelines) to enable automated parsing of the updated prompt.

Both meta-prompts are held fixed across all experiments; the only variables are the model-generated content passed through the placeholders (e.g., {cur_prompt}, {feedback}, {example_string}).

Feedback Collection Prompt. Below we present the prompt m_{fb} to collect structured feedback from $\mathcal{M}_{opt}$.

```
I am writing event guidelines and prompt (or task instructions) for a language model
     designed for an event extraction task.

My current prompt is:
<START>
{cur_prompt}
<END>

The event guideline in Python format is as following:
<START>
{event_definitions}
<END>

The task involves:
1. Extracting structured events (triggers, event type, arguments, and their roles)
    from the text.
2. Adhering to strict Python syntax for output (a Python list of event instances).
3. Handling all event definitions accurately, including mandatory roles and edge
    cases.

But this prompt gets the following examples wrong:
<START>
{example_string}
<END>

For each example, perform the following step-by-step analysis:
1. Error Type Classification: Identify the specific type(s) of error for each
    example (e.g., incorrect span extraction, missing roles, spurious arguments,
    format violations, etc.).
2. Root Cause Analysis:
    a. Did the current guideline fail to explain key extraction rules clearly?
    b. Are the instructions after `#` in the event definitions (guidelines)
        ambiguous, inconsistent, or insufficient?
    c. Were there ambiguities or overlaps in roles (e.g., `agent` vs. `person`) that
         caused confusion?
3. Example-Specific Recommendations:
    - Suggest precise changes to the guidelines (comments after `#` in event
        guidelines) to fix the errors for the given example.
    - Include explicit "what␣to␣do" and "what␣not␣to␣do" instructions for ambiguous
        roles or edge cases.
    - Provide a simple example and counterexample to illustrate each guideline.
4. General Trends: Identify recurring issues in guidelines across all examples.

Expected Output:
1. For all the examples, summarize and list all actionable changes to improve the
    event definitions for all the classes, including:
    - Improved clarity for event/role definitions.
    - Enhanced handling of ambiguous or overlapping roles.
    - Guidelines for precise span extraction.

2. Provide an output pointing out the mistakes in the current guidelines and propose
     refinements for all the classes. Each refinement should include:
    - For an event, updated guidelines for "what␣to␣do" and "what␣not␣to␣do."
    - Examples and counterexamples for each role.
```

Task Instruction and Guidelines Optimization Prompt. Below we present the prompt m_{opt} to optimize task instruction and event guidelines.

```
I am optimizing prompts for a language model designed for an event extraction task.

My current prompt (or task instructions) is:
<START>
{cur_prompt}
<END>

The event guideline in Python format is as following:
<START>
{event_definitions}
<END>

But this prompt gets the following examples wrong:
<START>
{example_string}
<END>

Based on these errors, the problems with the event guideline and the reasons are:
<START>
{feedback}
<END>

There are a list of former event guidelines including the current one, and each
    guideline is modified from its former prompts:
<START>
{trajectory_prompts}
<END>

Guidelines given to me for optimization of event classes:
1. Refine the prompt (or the task instructions) to address the issues mentioned
    previously. Focus on:
    - Clearer instructions for span extraction and role definitions along with any
        exceptions.
    - Handling ambiguous or overlapping roles effectively.
    - Strict adherence to Python-parsable output format.
2. Refine the guidelines for event definitions (the instructions after `#`) based on
     the identified mistakes. Ensure the refined guidelines addresses the concerns
    mentioned in the above.
3. Maintain backward compatibility: Ensure previously correct examples remain valid.
4. DO NOT change the ontology (Python classes). Instead, provide the refined
    guidelines in the format given at the end.
5. Ensure outputs follow these formats:
    - Optimized prompt (or the task instructions) wrapped with <START> and <END>.
    - Refined guidelines wrapped with <CLASS_START> and <CLASS_END>.

Output Requirements:
1. I have to provide the optimized prompt (or the task instructions) that evolves
    incrementally from the current one.
2. I also have to provide an output containing the fully optimized guidelines for
    each event definitions following the structure below:
class Event_Name(Parent_Event):
    \"\"\"
    # Updated guidelines here consulting the problems given to me
    \"\"\"
    mention: str # refined comments or extraction rules for event triggers. Include
    what/who can play the role with examples.
    {{role1}}: List # do the same for all roles including "mention", refining the
    comments after "#". Include what/who can play the role with examples and span
    extraction rule.


My response is:
```

E Optimized Task Instructions and Guidelines

In this section, we present fully optimized task instruction and event guidelines generated by DeepSeek-R1, o1, GPT-4.5, and GPT-4o.

E.1 Example of optimal task instruction and event guidelines generated by DeepSeek-R1

```
# Event Extraction Task: Extract structured events from text using Python class
    definitions. Follow these rules:

1. **Span Extraction**:
   - **Triggers**: Minimal contiguous spans (verbs/nouns) directly expressing the
       event. Include both verbal and nominal forms ("death" = Die, "killings" = Die
       ). Add new triggers like "converge" for Meet and "is␣no␣more" for EndOrg
   - **Arguments**:
     - Remove articles ("a/an/the") and possessive pronouns EXCEPT when part of
         official names or temporal phrases ("The␣Hague", "the␣past␣year")
     - Resolve pronouns AND POSSESSIVE NOUNS to named entities **immediately** using
          same-sentence antecedents ("airline's␣plan" → ["airline"])
     - Strip role/location/age descriptors from arguments ("Philadelphia␣lawyers" →
         "lawyers") unless part of multi-word crime
     - Keep FULL spans for crimes/money including sources/amounts ("stereo␣worth␣$1
         ,750␣from␣family") unless legal terms
     - Detect beneficiaries via ownership markers ("for␣X's␣project"), direct "to␣X"
          transfers go to recipient

2. **Special Handling**:
   - **Bankruptcy Triggers**: "went␣bust" → EndOrg unless explicit bankruptcy
       context
   - **Meet Entities**: Include ALL resolvable participants (subject + object)
   - **Crime Spans**: Retain full contextual clauses ("If␣convicted␣of␣killings...")
        without truncation
   - **Temporal Phrases**: Keep original spans with articles when part of phrase ("
       the␣early␣90's")

3. **Output Rules**:
   - Always output in Python-format as [EventName("mention" = "trigger", "arg1_key"
       = "arg1_span", ...), EventName("mention" = "trigger", "arg1_key" = "arg1_span
       ", ...)]
   - Include ALL role fields with empty lists where applicable
   - Output separate events for each trigger (no merging) even for identical event
       types
   - Strict pydantic syntax: [EventName(mention="span", role=["span"], ...)]
   - Preserve original casing for locations unless explicitly proper nouns

4. **Critical Exceptions**:
   - **EndOrg Triggers**: Add "collapse", "drive␣out", "went␣bust" with explicit org
        mentions
   - **Appeal Roles**: defendant = opposing party (state), prosecutor = appellant
   - **TransferMoney**: "for␣X" → recipient unless ownership marker ("for␣X's␣Y" →
       beneficiary)
   - **PhoneWrite Entities**: Strip ALL role descriptors ("Secretary␣Powell" → ["
       Powell"])



# Here are the event definitions:

class Convict(JusticeEvent):
    """Extract convictions where entity is found guilty of crime.
    Key Updates:
    - crime: Retain FULL spans including amounts/sources ("received stereo worth $1
        ,750 from family")

    Example: "convicted of taking bribes worth $1M" → crime=["taking bribes worth
        $1M"]
    Counterexample: Truncating to ["taking bribes"] → error
    """
    mention: str  # Triggers: "convicted", "conviction"
```

```
    defendant: List[str]  # ["Vang"] (resolved pronouns, strip descriptors)
    adjudicator: List[str]  # ["court"] (official names only)
    crime: List[str]  # Full offense span without legal terms
    time: List[str]  # ["last Wednesday"] (exact temporal phrases)
    place: List[str]  # ["Minnesota"] (geopolitical entities from context)

class TransferMoney(TransactionEvent):
    """Money transfers without goods exchange.
    Key Updates:
    - recipient: Direct receiver ("to X" OR "for X" if X is endpoint)
    - beneficiary: Only for ownership ("for X's project") or indirect benefit

    Example: "donated $5 for Tim Kaine" → recipient=["Tim Kaine"]
    Example: "funds for Kaine's campaign" → beneficiary=["Kaine"]
    """
    mention: str  # Triggers: "provided money", "donation"
    giver: List[str]  # ["foundation"] (strip descriptors)
    recipient: List[str]  # ["charity"] (direct receiver from "to/for X")
    beneficiary: List[str]  # ["Suha"] (from ownership markers)
    money: List[str]  # ["$15M"] (keep symbols/approximations)
    time: List[str]  # ["two years"] (full temporal span)
    place: List[str]  # ["Swiss"] (origin locations, strip prepositions)

class Meet(ContactEvent):
    """Face-to-face interactions.
    Key Updates:
    - entity: Include ALL resolvable participants (subject + object)

    Example: "Annan met Al-Douri" → entity=["Annan", "Al-Douri"]
    Counterexample: Omitting subject → error
    """
    mention: str  # Triggers: "meet", "summit", "talks"
    entity: List[str]  # ["delegates"] (all participants)
    time: List[str]  # ["today"] (exact temporal span)
    place: List[str]  # ["Dallas"] (resolved location noun)


class PhoneWrite(ContactEvent):
    """Non face-to-face communication.
    Key Updates:
    - entity: Strip ALL role descriptors unless part of compound name

    Example: "e-mail from Secretary Powell" → entity=["Powell"]
    Counterexample: Retaining "Secretary" → error
    """
    mention: str  # Triggers: "called", "e-mail" with transmission context
    entity: List[str]  # ["we", "them"] (bare names, resolved pronouns)
    time: List[str]  # ["during meeting"] (exact time phrase)
    place: List[str]  # ["office"] (specific location if present)


class DeclareBankruptcy(BusinessEvent):
    """Formal bankruptcy declarations.
    Key Rules:
    - entity: Resolve org pronouns AND possessive nouns ("airline's bankruptcy" → ["
        airline"])
    - Triggers: "bankruptcy", "Chapter 11" (exclude "collapse"/"went bust" without
        explicit bankruptcy context)

    Example: "airline's bankruptcy filing" → mention="bankruptcy", org=["airline"]
    Counterexample: "near-collapse" → EndOrg
    """
    mention: str  # Triggers indicating financial collapse: "bankruptcy", "Chapter
        11"
    entity: List[str]  # ["Enron Corp"] (resolved orgs from pronouns/possessives in
        same sentence)
    time: List[str]  # ["2003"] (declaration time phrase)
    place: List[str]  # ["Texas"] (jurisdiction noun if specified)

class EndOrg(BusinessEvent):
```

```
    """Organization termination events.
    Key Rules:
    - Triggers: "ceased", "is no more", "collapse", "drive out", "went bust"
    - org: Require explicit organizational mention ("casinos" in "casinos faced
        collapse")

    Example: "company went bust" → mention="went bust", org=["company"]
    Counterexample: "facing collapse" (no explicit org) → ignore
    """
    mention: str  # Triggers must indicate actual termination
    org: List[str]  # ["plant"] (direct object or possessive noun)
    time: List[str]  # ["the past year"] (with articles when part of phrase)
    place: List[str]  # ["Eugene"] (specific location noun)

class Die(LifeEvent):
    """Death events.
    Key Updates:
    - mention: Include nominal forms ("killings", "casualties") as valid triggers

    Example: "massacre casualties" → mention="casualties"
    Counterexample: "death penalty" → ignore
    """
    mention: str  # Triggers: "died", "killings", "casualties"
    agent: List[str]  # ["shooter"] (intentional actors only)
    victim: List[str]  # ["patient"] (without quantifiers/possessives)
    instrument: List[str]  # ["knife"] (specific tools/weapons)
    time: List[str]  # ["last night"] (exact span)
    place: List[str]  # ["hospital"] (death location noun)

class SentenceAct(JusticeEvent):
    """Punishment issuance events.
    Key Updates:
    - crime: Retain original crime from conditional clauses ("If convicted of
        killings..." → ["killings"])

    Example: "faces life for fraud" → crime=["fraud"]
    Counterexample: "could face penalty" → ignore
    """
    mention: str  # Triggers: "sentenced", "faces". Must reference actual punishment
    defendant: List[str]  # ["activist"] (strip role descriptors)
    adjudicator: List[str]  # ["jury"] (bare roles unless official title)
    crime: List[str]  # ["illegally attending meeting"] (full contextual span)
    sentence: List[str]  # ["life in prison"] (exact punishment phrase)
    time: List[str]  # ["Thursday"] (exact temporal expression)
    place: List[str]  # ["district court"] (decision location noun)

class Sue(JusticeEvent):
    """Legal action initiations.
    Key Updates:
    - adjudicator: Include "judge" if overseeing case approval ("approved by judge"
        → ["judge"])

    Example: "suit against Gateway approved by judge" → adjudicator=["judge"]
    Counterexample: "lawsuit documents" → adjudicator=[]
    """
    mention: str  # Triggers: "suit", "lawsuit". Must reference legal filing
    plaintiff: List[str]  # ["patients"] (strip locations/roles unless critical)
    defendant: List[str]  # ["Gateway"] (explicitly sued entities)
    adjudicator: List[str]  # ["judge"] (if directly involved)
    crime: List[str]  # ["malpractice"] (explicit offense without legal terms)
    time: List[str]  # ["last month"] (keep articles in temporal phrases)
    place: List[str]  # ["South Florida"] (specific noun phrases)

class Appeal(JusticeEvent):
    """Court decision appeals.
    Key Updates:
    - defendant: Opposing party (state/prosecution), NOT appellant
    - prosecutor: Entity filing appeal (resolved from subject/pronouns)
```

```
    Example: "appeal by Anwar against conviction" → prosecutor=["Anwar"], defendant
        =[]
    Counterexample: Assigning appellant as defendant → error
    """
    mention: str  # Triggers: "appeal", "appeals"
    defendant: List[str]  # ["state"] (opposing party in original case)
    prosecutor: List[str]  # ["Pasko"] (appellant, bare name without roles)
    adjudicator: List[str]  # ["court"] (original court name)
    crime: List[str]  # ["espionage"] (original charge)
    time: List[str]  # ["last week"] (exact temporal phrase)
    place: List[str]  # ["Malaysia"] (country from court description)
```

E.2 Example of optimal task instruction and event guidelines generated by o1

```
# This is an event extraction task where the goal is to extract structured events
    from the text following structured event definitions in Python.
# A structured event contains:
#   (1) an event trigger word (mention) -- always use the minimal lexical span (e.g
    ., "appeal" rather than "filed an appeal"),
#   (2) an event type, and
#   (3) the arguments participating in the event (with their roles).

# Keep argument references minimal by removing articles, possessives, or descriptive
     words unless they are crucial identifiers (e.g., "the retailer" -> "retailer",
    "my uncle" -> "uncle").

# Important guidelines to address prior errors:
#   1. For each event trigger, use the single most relevant word (e.g., "bankruptcy"
     rather than "file for bankruptcy").
#   2. For argument roles, also use minimal spans (e.g., "soldier" instead of "a
    soldier," "woman" instead of "a woman").
#   3. Output a separate event for each distinct trigger or implied event (e.g., a
    conviction and a subsequent sentencing should be two events).
#   4. For justice events (Sue, Appeal, Convict, SentenceAct, etc.):
#       - "defendant" is the party or entity accused or found guilty.
#       - "plaintiff" or "prosecutor" is the party initiating legal action or
    bringing an appeal. If the text does not specify who is accused, leave "
    defendant" empty.
#       - If the text refers to a punishment or sentencing (e.g., "faces the death
    penalty"), include a separate SentenceAct event referencing the same "defendant
    ."
#   5. For transfers of money, watch for direct or indirect references to donations,
     funding, or contributions and label them as TransferMoney events.
#   6. Do not skip events implied by synonyms or indirect wording (e.g., "shutting
    down" → EndOrg, "emerged from bankruptcy" → DeclareBankruptcy).
#   7. If there is more than one event in a single text, output each in a separate
    entry.
#   8. Always produce valid Python list format exactly as:
#       result = [
#         EventName("mention" = "trigger", "role1" = [...], "role2" = [...], ...),
#         EventName("mention" = "trigger", "role1" = [...], "role2" = [...], ...),
#       ]
#   9. Do not output anything else except this parsable Python structured format (no
     extra text or explanation).

# The event class definitions remain the same, but refer to the following refined
    docstrings for usage examples, minimal spans, and role clarifications.


# Here are the event definitions:

class Convict(JusticeEvent):
    """
    A Convict Event occurs whenever a Try Event ends with a successful prosecution
        of the Defendant.
    In other words, a Person, Organization or GPE Entity is convicted whenever that
        Entity has been
    found guilty of a Crime.
```

```
    Refined Guidelines:
      • mention: Use the minimal trigger word referring to the conviction (e.g., "
          guilty", "convicted").
      • defendant: The entity/ies found guilty. Remove articles or possessives ("the
           man" → "man").
      • adjudicator: The court or judge that issued the guilty verdict, if
          explicitly given.
      • crime: The wrongdoing for which the defendant was found guilty (e.g., "
          murdering X").
      • time: Any explicit time references (e.g., "last week").
      • place: Any explicit location references (e.g., "in Boston").

    What to do:
      - Include "crime" if stated: e.g., "convicted of murdering his wife" → crime
          =["murdering his wife"].
      - Keep the defendant arg minimal: "Scott Peterson" → ["Scott Peterson"], not
          ["Mr. Scott Peterson"].

    What not to do:
      - Do not guess or infer the crime if not stated.
      - Do not prepend articles or descriptive words (e.g., "the defendant" → "
          defendant" if used generically).

    Example:
      Text: "John was found guilty of fraud."
     → Convict(mention='guilty', defendant=['John'], crime=['fraud'], time=[],
          place=[])
    """
    mention: str  # minimal word expressing the conviction event
    defendant: List[str]  # who is found guilty
    adjudicator: List[str]  # the judge or court, if stated
    crime: List[str]  # the wrongdoing for which the defendant is convicted
    time: List[str]  # when the conviction takes place
    place: List[str]  # where the conviction takes place


class TransferMoney(TransactionEvent):
    """
    TransferMoney Events refer to giving, receiving, borrowing, or lending money
    when not purchasing goods or services in return.

    Refined Guidelines:
      • mention: Single word that triggers the transfer event (e.g., "donated", "
          loaned").
      • giver: The agent who provides funds. Remove determiners ("the", "a") unless
          part of a name.
      • recipient: The agent who receives the funds.
      • beneficiary: Any additional agent that benefits, if separate from recipient.
      • money: The amount of funds (if any mention like "$3,000", "large sum").
      • time: When the event takes place (e.g., "today", "last year").
      • place: Where the transaction or transfer occurs.

    What to do:
      - Label intangible references (e.g., "contributed", "had contributors") as
          TransferMoney if it implies funds.
      - Use minimal references for all money roles.

    What not to do:
      - Do not label intangible help (e.g., "emotional support") as TransferMoney.
      - Avoid listing indefinite articles or extraneous descriptors in the agent
          spans.

    Example:
      Text: "He donated $5,000 to Red Cross last week."
     → TransferMoney(mention='donated', giver=['He'], recipient=['Red Cross'],
          money=['$5,000'], time=['last week'], place=[])
    """
    mention: str  # minimal word triggering the money transfer
    giver: List[str]  # who provides the money
    recipient: List[str]  # who receives the money
```

```
    beneficiary: List[str]  # who additionally benefits, if any
    money: List[str]  # the sum or amount
    time: List[str]  # when the transfer happens
    place: List[str]  # where the transfer event occurs


class Meet(ContactEvent):
    """
    A Meet Event occurs when two or more Entities come together face-to-face
    at a single location and interact with one another.

    Refined Guidelines:
      • mention: The single best word for the meeting (e.g., "met", "summit", "
          conference").
      • entity: All participants, stripped of articles or descriptors. If multiple,
          list them all.
      • time: Any temporal phrase referencing when the event took place.
      • place: The location of the meeting.

    What to do:
      - Use triggers for in-person gatherings (e.g., "met", "conference", "summit").
      - Keep participant references minimal: "President", "Vice-President" instead
          of "the US President".

    What not to do:
      - Do not treat phone calls or written communication as Meet (use PhoneWrite).

    Example:
      Text: "The leaders met in Paris yesterday."
      → Meet(mention='met', entity=['leaders'], time=['yesterday'], place=['Paris'])
    """
    mention: str  # minimal word or short phrase for the meeting
    entity: List[str]  # who met face-to-face
    time: List[str]  # when the meeting happened
    place: List[str]  # where the meeting occurred


class PhoneWrite(ContactEvent):
    """
    A PhoneWrite Event occurs when two or more people communicate
    without meeting face-to-face. This includes phone calls, email, texting, etc.

    Refined Guidelines:
      • mention: The minimal expression of communication (e.g., "called", "emailed",
          "texted").
      • entity: The agents communicating. Strip out articles, determiners, or extra
          descriptors.
      • time: When the communication took place (e.g., "this morning", "yesterday").

    What to do:
      - Common triggers: "phoned", "emailed", "talked by phone", "texted", "messaged
          ".
      - Keep roles minimal (e.g., entity=['John', 'Mary']).

    What not to do:
      - Do not mark in-person discussions as PhoneWrite (use Meet).

    Example:
      Text: "They emailed each other last night."
      → PhoneWrite(mention='emailed', entity=['They'], time=['last night'])
    """
    mention: str  # minimal communication trigger
    entity: List[str]  # communicating parties
    time: List[str]  # when the communication happened


class DeclareBankruptcy(BusinessEvent):
    """
    A DeclareBankruptcy Event occurs whenever an Entity officially seeks legal
        protection
```

```
    from debt collection due to severe financial distress.

    Refined Guidelines:
      • mention: Short trigger related to bankruptcy (e.g., "bankruptcy", "filed", "
          declared").
      • org: The organization or person who declares bankruptcy. Remove "the", "my",
           etc.
      • time: When the bankruptcy is declared (e.g., "in 2003", "today").
      • place: Where the declaration is made, if mentioned (e.g., "in court", "in
          New York").

    What to do:
      - Recognize synonyms or indirect references like "emerged from bankruptcy" or
          "bankruptcy protection" as triggers.

    What not to do:
      - Do not guess an org if not specified.

    Example:
      Text: "My uncle declared bankruptcy in 2003."
     → DeclareBankruptcy(mention='bankruptcy', org=['uncle'], time=['2003'], place
          =[])
    """
    mention: str  # minimal expression for bankruptcy
    org: List[str]  # the party declaring bankruptcy
    time: List[str]  # when the declaration takes place
    place: List[str]  # where it is declared


class EndOrg(BusinessEvent):
    """
    An EndOrg Event occurs when an Organization ceases to exist or
    "goes out of business."

    Refined Guidelines:
      • mention: Minimal trigger (e.g., "shutting down", "closing").
      • org: The organization or sub-unit that ends. E.g., "plant", "branch".
      • time: When this closure or end is stated to happen.
      • place: Where the organization is located or ended.

    What to do:
      - Consider references such as "closing its plant" → "plant" in org.
      - Identify synonyms like "shutting down," "ceasing operations."

    What not to do:
      - Do not skip it if the text explicitly says the org ended.

    Example:
      Text: "Hewlett Packard is shutting down its plant in Eugene."
     → EndOrg(mention='shutting down', org=['plant'], time=[], place=['Eugene'])
    """
    mention: str  # minimal expression for the organizational end
    org: List[str]  # the ended organization
    time: List[str]  # when the end occurs
    place: List[str]  # where this event happens


class Die(LifeEvent):
    """
    A Die Event occurs whenever a Person loses their life, whether accidental,
    intentional, or self-inflicted.

    Refined Guidelines:
      • mention: The short trigger referencing the death (e.g., "killed", "died", "
          murdered").
      • agent: The killer or cause if identified (e.g., "gunman", "regime")-remove
          articles.
      • victim: Who died, again with minimal references (e.g., "soldier" instead of
          "a soldier").
      • instrument: The device or method used, if any (e.g., "gun", "bomb").
```

```
      • time: When the death occurred.
      • place: Where it took place.

    What to do:
      - Create separate Die events for each death trigger in the text.
      - If the text references homicide: agent is the killer, victim is the deceased
          .

    What not to do:
      - Do not combine multiple victims into one string if they appear as separate
          triggers.

    Example:
      Text: "He killed the soldier in Iraq."
      → Die(mention='killed', agent=['He'], victim=['soldier'], instrument=[], time
          =[], place=['Iraq'])
    """
    mention: str  # minimal word referencing the death
    agent: List[str]  # optional killer or cause
    victim: List[str]  # who died
    instrument: List[str]  # how they were killed (weapon, etc.)
    time: List[str]  # when the death happened
    place: List[str]  # where the death happened


class SentenceAct(JusticeEvent):
    """
    A SentenceAct Event occurs whenever a punishment for the Defendant is issued,
    e.g., a prison term or another legal penalty.

    Refined Guidelines:
      • mention: A trigger referencing sentencing or punishment (e.g., "sentenced",
          "faces [penalty]").
      • defendant: The same party convicted or found guilty, if known.
      • adjudicator: The entity delivering the sentence, if stated (e.g., "judge", "
          court").
      • crime: The wrongdoing for which the defendant is sentenced (e.g., "murder",
          "embezzlement").
      • sentence: The specific punishment (e.g., "death penalty", "life in prison").
      • time: When the sentencing occurs.
      • place: Where the sentencing occurs.

    What to do:
      - Look for words like "faces the death penalty," "was sentenced to ten years."

    What not to do:
      - Do not omit a SentenceAct if there's explicit mention of punishment.

    Example:
      Text: "He now faces the death penalty for murdering his wife."
      → SentenceAct(mention='faces', defendant=['He'], crime=['murdering his wife'],
          sentence=['death penalty'], time=[], place=[])
    """
    mention: str  # minimal expression for the sentencing event
    defendant: List[str]  # who is sentenced
    adjudicator: List[str]  # judge or court
    crime: List[str]  # the wrongdoing or offense
    sentence: List[str]  # the punishment
    time: List[str]  # when the sentencing happens
    place: List[str]  # where it happens


class Sue(JusticeEvent):
    """
    A Sue Event occurs whenever a court proceeding is initiated to determine
    the liability of a Person, Organization, or GPE.

    Refined Guidelines:
      • mention: The minimal trigger (e.g., "sued", "suing", "filed a lawsuit", "
          suit").
```

```
      • plaintiff: The party bringing the suit. Strip out any articles or adjectives
        .
      • defendant: The party being sued. Again, keep references minimal.
      • adjudicator: The judge or court if one is explicitly named.
      • crime: If a wrongdoing is stated (e.g., "for fraud", "for breach of contract
        ").
      • time: When the suit is filed or mentioned.
      • place: Where the suit is taking place.

    What to do:
      - Label the party initiating the lawsuit as "plaintiff."

    What not to do:
      - Do not confuse "plaintiff" with "defendant" if the text clearly states who
          is suing whom.

    Example:
      Text: "A nurse sued Dell for bait and switch."
      → Sue(mention='sued', plaintiff=['nurse'], defendant=['Dell'], crime=['bait
          and switch'], time=[], place=[])
    """
    mention: str  # minimal expression for the lawsuit event
    plaintiff: List[str]  # who brings the suit
    defendant: List[str]  # who is being sued
    adjudicator: List[str]  # the judge or court, if stated
    crime: List[str]  # the wrongdoing for which the suit is filed
    time: List[str]  # when the suit took place
    place: List[str]  # where the suit took place


class Appeal(JusticeEvent):
    """
    An Appeal Event occurs whenever a court decision is taken to a higher court
    for review.

    Refined Guidelines:
      • mention: The short trigger for the appeal (e.g., "appeal", "appealed").
      • defendant: The party accused or found guilty, if the text states so.
      • prosecutor: The party bringing the appeal (i.e., the appellant). This might
          be the same individual who was a defendant in a prior trial but is now
          appealing.
      • adjudicator: The higher court or judge handling the appeal, if given.
      • crime: The wrongdoing for which the appeal is made (if stated).
      • time: When the appeal is filed or heard.
      • place: Where the appeal is taking place.

    What to do:
      - If text says someone "filed an appeal," that entity is the "prosecutor" if
          no other roles are specified.
      - If the text does not identify an accused, keep defendant=[].

    What not to do:
      - Do not automatically fill "defendant" if it's unclear who was accused.

    Example:
      Text: "He appealed the verdict last week."
      → Appeal(mention='appealed', defendant=[], prosecutor=['He'], crime=[], time
          =['last week'], place=[])
    """
    mention: str  # minimal word for the appeal event
    defendant: List[str]  # the accused, if stated
    prosecutor: List[str]  # who is bringing the appeal
    adjudicator: List[str]  # the judge or court for the appeal
    crime: List[str]  # the crime or issue being appealed
    time: List[str]  # when the appeal occurs
    place: List[str]  # where the appeal is heard
```

E.3 Example of task instruction and optimal event guidelines generated by GPT-4.5

```
# This is an event extraction task for identifying and structuring events from text
    using Python-defined event classes. Each structured event consists of an event
    trigger word, an event type, participant arguments, and their roles. Your
    objective is to output this information in a Python list of events, ensuring it
    is Python-parsable and strictly follows the event definitions provided below.

## Instructions:

1. **Span Extraction**:
    - Extract precise and concise spans for mentions and participant arguments,
        conveying the event or argument role clearly without unnecessary context.
    - For extracts involving titles or specifics, use general terms unless details
        are crucial to the events integrity.
    - When identifying entity roles in events, prioritize the core identifiers over
        accompanying descriptors.

2. **Role Identification**:
    - Accurately identify roles using contextual cues, effectively resolving
        ambiguities while prioritizing explicit spans. If roles are unmentioned,
        leave them empty.
    - Maintain consistency, particularly with distinctions like plaintiff vs.
        defendant, based on contextual evidence.
    - Clarify roles in complex transactions, such as distinguishing between
        beneficiaries and direct recipients.

3. **Output Format**:
    - Please follow the Python-format EventName("mention" = "trigger", "role1" =
        [...], "role2" = [...], ...) strictly.
    - Ensure consistent output in the specified format for Python compatibility,
        adhering strictly to event definitions.
    - Represent unmentioned participants with an empty list rather than assumptions
        or placeholders.

4. **Clarifications and Exceptions**:
    - Note explicitly when roles have exceptions based on role definitions.
    - Manage overlapping roles by following specific guidelines for span clarity and
         precision, ensuring no crucial details are overlooked.

5. **Consistency**:
    - Ensure consistency in role identification and event extraction across similar
        scenarios.
    - Address ambiguity and overlap by defining roles explicitly and setting clear
        precedence for extraction guidelines.

Below are the structured event definitions:




# Here are the event definitions:

class Convict(JusticeEvent):
    """
    A Convict Event signifies the successful prosecution of a defendant. This
        involves a person, organization, or geographical political entity (GPE)
        being convicted for a crime.
    """
    mention: str  # Focus on concise triggers like "convicted" or "conviction",
        avoiding embellishments.
    defendant: List[str]  # Name the convicted individuals or entities. Use direct
        identifiers, example: "John Doe".
    adjudicator: List[str]  # Reference the judicial entity, example: "court" or "
        judge", unless specifics are critical.
    crime: List[str]  # Provide short, precise descriptions of crimes, e.g., "fraud
        ".
    time: List[str]  # Specify exact times if mentioned, e.g., "Monday".
    place: List[str]  # Note locations if explicitly mentioned, avoid assumptions.

class TransferMoney(TransactionEvent):
```

```
    """
    Non-purchasing money transfers involving giver and recipient roles, where
        transactions are more indirect or complex.
    """
    mention: str  # Use explicit terms like "donated", staying concise.
    giver: List[str]  # Identify the money source, example: "Sheila C. Johnson".
    recipient: List[str]  # Clearly name receiving entities.
    beneficiary: List[str]  # Note additional beneficiaries unambiguously.
    money: List[str]  # Use exact figures, avoiding vague amounts.
    time: List[str]  # Define occurrence times if clearly specified.
    place: List[str]  # Mention the transaction location if detailed.

class Meet(ContactEvent):
    """
    Events where entities gather face-to-face, e.g., meetings, summits, or
        conferences.
    """
    mention: str  # Central meeting references like "summit", without extra detail.
    entity: List[str]  # List participants clearly, omitting superfluous
        descriptions.
    time: List[str]  # Specify times if explicitly provided.
    place: List[str]  # Mention locations if available, avoiding unsupported
        assumptions.

class PhoneWrite(ContactEvent):
    """
    Non-face-to-face communications, covering written and phone-based interactions.
    """
    mention: str  # Terms indicating communication, e.g., "called", succinctly.
    entity: List[str]  # Capture the participants in the communication.
    time: List[str]  # Specify times if mentioned, ensuring clarity.

class DeclareBankruptcy(BusinessEvent):
    """
    Occurs when an organization requests legal protection from debt collection.
    """
    mention: str  # Use declarations like "bankruptcy", clearly.
    org: List[str]  # Focus on the organizational name in question.
    time: List[str]  # Mention when the declaration occurs if explicitly stated.
    place: List[str]  # Note the declaration's location if outlined.

class EndOrg(BusinessEvent):
    """
    An organization ceases operations, going out of business completely.
    """
    mention: str  # Use terms like "shut down" to capture essence effectively.
    org: List[str]  # Succinctly list the organizations ending operations.
    time: List[str]  # Clearly mention when specifics are supplied.
    place: List[str]  # Mention location details if clearly stated.

class Die(LifeEvent):
    """
    Event marking the end of life, covering direct, accidental, and self-inflicted
        cases.
    """
    mention: str  # Specific terms like "died", excluding excess context.
    agent: List[str]  # Cite any responsible party if indicated.
    victim: List[str]  # Precisely identify the deceased without titles.
    instrument: List[str]  # Specify instruments used if described.
    time: List[str]  # Use accurate timing where provided.
    place: List[str]  # Mention locations where explicitly noted.

class SentenceAct(JusticeEvent):
    """
    Legal sentence issuance, often involving incarceration.
    """
    mention: str  # Direct words like "sentenced", retaining clarity.
    defendant: List[str]  # Identify the sentenced party succinctly.
    adjudicator: List[str]  # State the authority issuing the sentence.
    crime: List[str]  # Precisely include mentioned crimes.
```

```
    sentence: List[str]  # Clearly outline the penalties involved.
    time: List[str]  # Specific timing if explicitly declared.
    place: List[str]  # Cite location details when supplied.

class Sue(JusticeEvent):
    """
    The initiation of legal proceedings against an entity to determine liability.
    """
    mention: str  # Specific terms like "sued".
    plaintiff: List[str]  # Clearly identify the suing parties.
    defendant: List[str]  # Identify the sued entities unambiguously.
    adjudicator: List[str]  # Specify judicial role if expressed.
    crime: List[str]  # Highlight alleged crimes if specified.
    time: List[str]  # Reference explicit timing if detailed.
    place: List[str]  # Extract the location details if outlined.

class Appeal(JusticeEvent):
    """
    Represents decisions moved to higher courts for further review.
    """
    mention: str  # Use terms like "appealed" directly.
    defendant: List[str]  # Name the entity under review.
    prosecutor: List[str]  # Name the initiating party of the appeal.
    adjudicator: List[str]  # Reference the reviewing court.
    crime: List[str]  # Clearly detail crimes if mentioned.
    time: List[str]  # Capture filing times if explicit.
    place: List[str]  # Mentioned locale of appeal if detailed.
```

E.4 Example of optimal task instruction and event guidelines generated by GPT-4o

```
# This is an event extraction task where the goal is to extract structured events
    from the text following structured event definitions in Python. A structured
    event contains an event trigger word, an event type, the arguments participating
     in the event, and their roles in the event.

# Task Instructions:
1. For each different event type, output the extracted information from the text
    into a Python list format where:
   - The first key 'mention' holds the value of the event trigger.
   - Subsequent keys/values follow the class definitions below.

2. Structure the output in a valid Pydantic format: `result = [EventName("mention" =
     "trigger", "arg1_key" = "arg1_span", ...)]`.
3. Adhere strictly to the described event descriptions and role definitions,
    considering implicit contexts and indirect attributions.
4. Address special cases:
   - Appeals: Consider involved parties from prior related events as “prosecutor”.
   - Multiple roles may apply contextually; ensure complete information extraction.
   - Implicit indications: If mentions like "filed", "concluded", etc., suggest
       indirect roles, use context to clarify them.

5. Maintain backward compatibility where applicable. Do not output anything else
    except parsable structured event format in Python.





# Here are the event definitions:

class Convict(JusticeEvent):
    """
    A Convict Event occurs whenever a Try Event ends with a successful prosecution
        of the Defendant.
    There may not always be explicit mentions of crimes in the text; use contextual
        clues.
    """
    mention: str  # The text span that expresses the conviction (e.g., "convicted").
```

```
    defendant: List[str]  # The entity found guilty, search for adjacent terms like
        "defendant".
    adjudicator: List[str]  # The judge or court, often implicitly understood from
        context.
    crime: List[str]  # Crime references, even implied (e.g., "guilty of ...").
    time: List[str]  # When conviction happens, contextual or explicit dates.
    place: List[str]  # Where the conviction occurs, often a court or city name
        nearby.

class TransferMoney(TransactionEvent):
    """
    Refers to money transfer actions outside purchasing contexts. Recognize givers
        and recipients even in indirect mentions.
    """
    mention: str  # Turn of phrase indicating transfer (e.g., "transferred", "
        donated").
    giver: List[str]  # Entity initiating transfer (may be implied; use context).
    recipient: List[str]  # Direct receiver of money, often clearly stated.
    beneficiary: List[str]  # Can be implied; beneficiaries are often indirect.
    money: List[str]  # Described amounts; look for currency signs ($, €, etc.).
    time: List[str]  # Dates or relative times (e.g., "two years ago").
    place: List[str]  # Locations of transaction, if specified.

class Meet(ContactEvent):
    """
    Occurs when entities meet face-to-face; discern collective entity mentions from
        individual roles.
    """
    mention: str  # Trigger phrases (e.g., "meet", "conference").
    entity: List[str]  # Entities, clarified through context or explicit mentions.
    time: List[str]  # When entities meet, even if future planned.
    place: List[str]  # Meeting location, from nearby phrases.

class PhoneWrite(ContactEvent):
    """
    Encompasses non-face-to-face communications; cover implied interactors.
    """
    mention: str  # Non-direct communication identified triggers (e.g., "called", "
        emailed").
    entity: List[str]  # Communicating entities, occasionally understood indirectly.
    time: List[str]  # Times derived from text, even if not very specific.

class DeclareBankruptcy(BusinessEvent):
    """
    An event signifying financial distress declarations; distinguish from emergence
        narratives.
    """
    mention: str  # Indicators like "declared bankruptcy".
    org: List[str]  # Company/entity that declared, directly mentioned.
    time: List[str]  # Declaration date, often provided.
    place: List[str]  # Geographical context of declaration.

class EndOrg(BusinessEvent):
    """
    Concludes an organization's operations; ensure specificity of organization
        ceases.
    """
    mention: str  # Marks of closure (e.g., "dissolved", "shutdown").
    org: List[str]  # Organization ending, referenced in texts.
    time: List[str]  # Date context around organization ending.
    place: List[str]  # Location tied to organizational operations.

class Die(LifeEvent):
    """
    Recognizes cessation of life events; determine involvements from surrounding
        text.
    """
    mention: str  # Triggering term showing death (e.g., "died", "passed away").
    agent: List[str]  # Agents causing death if deliberate; contextual deductions.
    victim: List[str]  # Deceased, named or implied victims.
```

```
    instrument: List[str]  # Weapons or causes if mentioned.
    time: List[str]  # Death-related timing, even metaphorical.
    place: List[str]  # Place the death occurred, discerned from text.

class SentenceAct(JusticeEvent):
    """
    Legal actions culminating in punishment; include implied authority adjudication
        references.
    """
    mention: str  # Verbs indicating sentencing (e.g., "sentenced").
    defendant: List[str]  # Persons sentenced, more direct mentions.
    adjudicator: List[str]  # State actor issuing punishment.
    crime: List[str]  # Crimes specified can be explicit or by context related.
    sentence: List[str]  # Detailed punishments, commonly listed.
    time: List[str]  # Contextual timing of legal processes.
    place: List[str]  # Legal venues, stated or implicit.

class Sue(JusticeEvent):
    """
    Legal actions initiation detections; interpreting mentions to detect implicated
        parties.
    """
    mention: str  # Lawsuit trigger terms (e.g., "sued").
    plaintiff: List[str]  # Agents initiating, even implicit from context.
    defendant: List[str]  # Specific subjects of the lawsuit.
    adjudicator: List[str]  # Legal bodies, typically explicit.
    crime: List[str]  # Charges or offenses underpinning the suit.
    time: List[str]  # Suit filing and related timings.
    place: List[str]  # Locations cited, often courts.

class Appeal(JusticeEvent):
    """
    Reviewal legal challenges; correctly attribute events around appellate actions.
    """
    mention: str  # Terms denoting appeals like "appealed".
    defendant: List[str]  # Party whose case goes under review.
    prosecutor: List[str]  # Original case actors initiating the appeal, inferred.
    adjudicator: List[str]  # Higher court taking the over evaluation.
    crime: List[str]  # Reviews' subject offenses.
    time: List[str]  # Appeal reference times, may not be given.
    place: List[str]  # Court location details or broader judicial zones.
```