SELF-STEERING LANGUAGE MODELS

Gabriel Grand[1]*
Joshua B. Tenenbaum[1] **Vikash K. Mansinghka**[1]
Alexander K. Lew[1,2] **Jacob Andreas**[1]
[1]Massachusetts Institute of Technology [2]Yale University
`{gg, jbt, vkm, jda}@mit.edu` `alexander.lew@yale.edu`

ABSTRACT

While test-time reasoning enables language models to tackle complex tasks, searching or planning in natural language can be slow, costly, and error-prone. But even when LMs struggle to emulate the precise reasoning steps needed to solve a problem, they often excel at describing its *abstract structure*—both how to verify solutions and *how to search* for them. This paper introduces DISCIPL, a method for "self-steering" LMs where a *Planner model* generates a task-specific *inference program* that is executed by a population of *Follower models*. Our approach equips LMs with the ability to write recursive search procedures that guide LM inference, enabling new forms of verifiable and efficient reasoning. When instantiated with a small Follower (e.g., `Llama-3.2-1B`), DISCIPL matches (and sometimes outperforms) much larger models, including `GPT-4o` and `o1`, on challenging constrained generation tasks. In decoupling planning from execution, our work opens up a design space of highly-parallelized Monte Carlo inference strategies that outperform standard best-of-N sampling, require no finetuning, and can be implemented automatically by existing LMs.

1 INTRODUCTION

Even as language models (LMs) are becoming increasingly proficient reasoners, progress has been "jagged" (Karpathy, 2024; Roose, 2025): today's frontier models surpass experts at science and math reasoning (e.g., Hendrycks et al., 2021; Rein et al., 2023; Wang et al., 2024b) yet still routinely struggle with counting, arithmetic, tic-tac-toe, metered poetry, and other intuitively simple tasks (Ball et al., 2024; McCoy et al., 2023; Xu & Ma, 2024). For example, even very capable LMs have difficulty writing a coherent sentence under the constraints in Fig. 1, which are manageable for most proficient English speakers.

There is a growing consensus that many kinds of queries require more thoughtful, deliberate "System-2" reasoning. However, a key question is how best to leverage computation at test-time. One currently popular approach induces in-context reasoning via long chains-of-thought (DeepSeek-AI et al., 2025; OpenAI, 2024b). This approach is very flexible, allowing the LM to decide on a problem-by-problem basis how to structure its thinking. But reasoning via (serial) autoregressive generation is costly, slow, and can still produce unreliable outputs. On the other hand, structured inference methods like tree search (Silver et al., 2016; Yao et al., 2023) and sequential Monte Carlo (Lew et al., 2023; Loula et al., 2025; Zhao et al., 2024) attain better parallelism and efficiency by coordinating test-time computation via external algorithms. However, because these approaches typically require pre-defined scorers or verifiers, and rely on LMs to produce correct outputs with non-negligible probability, their applications have so far been restricted to specific domains.

In this work, we propose a new meta-reasoning framework called DISCIPL in which language models *themselves* drive the decisions for how to structure inference-time compute. In our approach, a **Planner LM** is fed the user's query and asked to generate an ad-hoc specification (encoding its understanding of the task requirements) and inference procedure (encoding its plan for how to solve the task). Importantly, this plan is implemented as an *inference program* that invokes **Follower LMs**, either generatively or as likelihood evaluators. By decomposing reasoning into planning and execution, our architecture preserves flexibility while enabling orchestration of highly efficient, parallel search patterns.

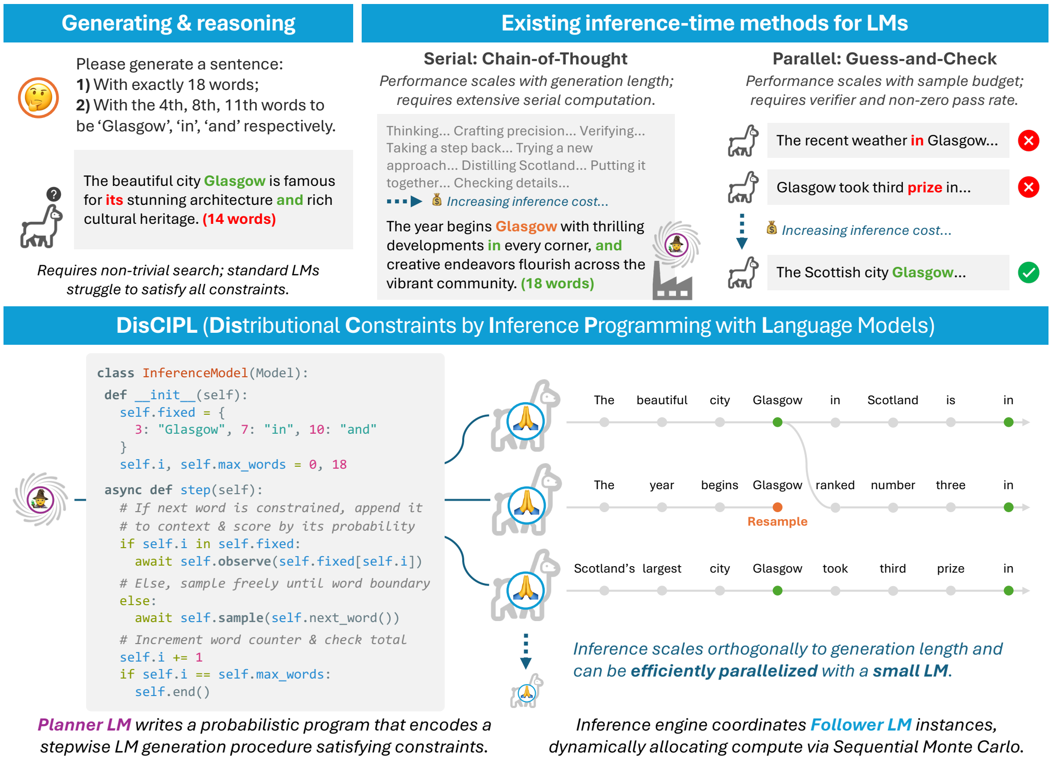

Figure 1: **Self-steering language models with probabilistic programs.** LMs struggle with problems that combine generation and reasoning under constraints (*top left*, task from Yao et al., 2024). Popular approaches (*top right*) scale inference by search or sampling; however, even costly reasoning models do not always yield correct or fluent results (CoT example from o1, single-shot, first attempt). In our method (DisCIPL, *bottom*), a Planner LM writes an *inference program* that defines step-by-step computations to steer a population of Follower LMs. Our approach combines the benefits of serial and parallel methods: the Planner ensures correctness by construction, while the Followers collectively search for sequences with high probability. (See Fig. 6 for a detailed visualization.)

To test this approach, we evaluate DisCIPL on two domains: (1) Collie (Yao et al., 2024), a challenging constrained generation benchmark on which even very large LMs perform unreliably; and (2) Puzzles, a custom dataset of difficult tasks involving poetry composition, grant-writing, budgeting, and itinerary planning. We instantiate DisCIPL using a capable Planner LM (`GPT-4o` prompted with few-shot examples) to generate code, and a small Follower LM (`Llama-3.2-1B`) to execute it. On paragraph tasks, this approach boosts accuracy well beyond the original capabilities of the Follower. Meanwhile, on more tightly constrained sentence and puzzle tasks, DisCIPL enables the Follower to surpass the Planner and approach the performance of powerful reasoning models like `o1`.

Our method builds on the growing body of recent work showing that the effectiveness of LMs can be dramatically improved with task-specific search and sampling methods. By using LMs to define these procedures on-the-fly, DisCIPL provides efficient, fully-automated solutions to problems that would otherwise require significant computational overhead or manual engineering effort. More broadly, this work offers a new way to combine code generation and probabilistic inference to enable test-time scaling for LMs.

2 Related Work

Scaling Inference-Time Computation Reasoning via autoregressive generation is effective for many tasks, and the success of "chain-of-thought" (Nye et al., 2021; Wei et al., 2023) has spawned many variants that trade off generality and efficiency: whereas constructing an explicit "tree-of-thought" (Liu et al., 2024; Yao et al., 2023) requires problem-specific engineering, linearizing reasoning into one long "stream-of-search" (Gandhi et al., 2024; Lehnert et al., 2024) scales exponentially with tree depth. As an alternative, simple sampling procedures like best-of-N sampling

(Brown et al., 2024; Cobbe et al., 2021) and self-consistency/majority voting (Wang et al., 2023a) have also emerged as a popular means of scaling LM performance. These "embarrassingly parallel" methods (Herlihy & Shavit, 2012) are simple to implement, but determining the optimal sample budget for a new problem requires special techniques to estimate expected task difficulty (i.e., adaptive sampling; Damani et al., 2024; Snell et al., 2024).

Self-Improvement. Recently, a variety of novel methods have been proposed for using LMs to optimize prompts (Fernando et al., 2023; Honovich et al., 2022; Khattab et al., 2023; Shinn et al., 2023; Yang et al., 2024; Zhou et al., 2023c), agentic systems (Hu et al., 2024), and optimization procedures themselves (Zelikman et al., 2024). DISCIPL shares a similar recursive flavor, but generates ad-hoc *inference algorithms* that provide token-level control over LM behavior at decoding time with probabilistic guarantees.

Constrained Generation. Recent years have seen a proliferation of benchmarks designed to test the ability of LMs to adhere to complex and compositional constraints (Chia et al., 2023; Jiang et al., 2023; Lin et al., 2020; Sun et al., 2023; Wang et al., 2022; Yao et al., 2024; Zhou et al., 2023a). Prior approaches have focused on developing decoding algorithms for specific classes of constraints (Hokamp & Liu, 2017; Koo et al., 2024; Lu et al., 2021; 2022; Poesia et al., 2022; Post & Vilar, 2018; Ugare et al., 2024; Willard & Louf, 2023) or guiding generation with neural models (Amini et al., 2024; Kumar et al., 2022; Li et al., 2022; Qin et al., 2022). Various learning-based approaches have also been proposed, including self-correction through feedback (Welleck et al., 2022) and bootstrapping from self-generated instructions (Wang et al., 2023b; Zhou et al., 2023b).

Sequential Monte Carlo with LMs. Particle-based methods, such as sequential Monte Carlo (SMC; Doucet et al., 2001), offer a powerful framework for building adaptive inference-time search procedures. SMC has recently been applied to LMs to solve various tasks, including constrained generation (Lew et al., 2023; Loula et al., 2025; Zhao et al., 2024) and math reasoning (Feng et al., 2025; Puri et al., 2025). However, using SMC on new problems requires either engineering or learning various parameters of the algorithm (e.g., reward models or twist functions); in our work, an LM automates this process.

Probabilistic Programming and LMs. Probabilistic programming languages (PPLs) allow users to implement probabilistic models as programs, and automate aspects of probabilistic inference (Goodman et al., 2014). Some PPLs support LMs as parts of models (Dohan et al., 2022; Lew et al., 2020; 2023), and LMs have also been used to generate probabilistic programs over symbolic domains (Li et al., 2024; Wong et al., 2023). More recently, PPLs have evolved to feature *programmable inference* (Mansinghka et al., 2014; 2018), so that programs can concisely specify both models and custom inference algorithms. We use LMs to generate code in LLAMPPL (Lew et al., 2023), a PPL built on a modern LM stack that enables users to define inference procedures via short, intuitive Python programs.

3 DISCIPL

3.1 GENERAL FRAMEWORK

A *language model* M is a distribution on token sequences from a vocabulary $\boldsymbol{x} \in \mathcal{V}^*$ whose probability factorizes autoregressively as $p_M(\boldsymbol{x}) = \prod_{t=1}^{|x|} p_M(x_t \mid \boldsymbol{x}_{<t})$. We designate roles for two LMs, a **Planner** M_P and a **Follower** M_F. We assume that both support efficient *conditional generation*, and that M_F also supports efficient *conditional probability evaluation* (i.e., access to logprobs).

The key idea in DISCIPL is that the Planner can generate an *inference program* π in some language $\mathcal{L}$ that describes how the Follower should be used to solve a task. The program may make multiple asynchronous queries to the Follower, in both generative (i.e., sampling) and evaluative (i.e., probability computation) modes. The language provides an *inference engine* $\mathcal{I}$ that takes a program $\pi \in \mathcal{L}$, a Follower model M_F, and an *inference budget* $N \in \mathbb{N}$, and yields samples from a distribution on results, which can either be *answers* $\boldsymbol{x} \in \mathcal{V}^*$ or *errors* $\varepsilon \in \mathcal{E}$.

With these ingredients, the basic DISCIPL algorithm (Alg. 1) proceeds as follows. Given a natural language task description d_{task}, we first prompt the Planner M_P to generate a program in $\pi \in \mathcal{L}$ suitable for solving the task. We then attempt to run π using the inference engine $\mathcal{I}$. If inference succeeds, we return its output. Otherwise, we re-prompt M_P to correct π, passing the error ε and associated traceback, up to a max number of retries R.

Algorithm 1 DISCIPL Outer Loop

function DISCIPL(*task* d_{task}, *Planner* M_P, *Follower* M_F, *budget* N, *retries* R, *engine* $\mathcal{I}$)
  result $\leftarrow \varepsilon_{\text{null}}, r \leftarrow 1$ ▷ *Initialize with empty string*
  while *result* $\in \mathcal{E}$ **and** $r \leq R$ **do**
    $\pi \sim p_{M_P}(\cdot \mid \texttt{prompt}(d_{\text{task}}, \textit{result}))$ ▷ *Generate inference program (with prior errors as feedback)*
    result $\sim \mathcal{I}(\pi, M_F, N)$ ▷ *Execute the program with Follower*
    if *result* $\in \mathcal{E}$ **then**
      $r \leftarrow r + 1$ ▷ *Retry on runtime errors*
  return *result*

3.2 OUR INSTANTIATION: LANGUAGE MODEL PROBABILISTIC PROGRAMS

Language of inference programs. In our instantiation of DISCIPL, programs π are written in LLAMPPL (Lew et al., 2023), a Python framework for probabilistic programming with language models. LLAMPPL programs work by repeatedly *extending* candidate generations by one or more tokens, and then *scoring* the proposed extensions. The programmer (in our case, the Planner model) can customize the process by which new extensions are proposed and scored; behind the scenes, LLAMPPL's inference engine automatically coordinates the overall search, maintaining multiple candidate generations in parallel, and dynamically reallocating computational resources to high-scoring partial completions.

As a concrete example, consider the program in Fig. 1, whose `step` method stochastically adds one word to the running generation, either sampling from M_F (using `sample`) or forcing it to agree with a word constraint (using `observe`). The `observe` method has the side effect of updating the candidate's *score*, multiplying it by the probability the Follower assigns to the observed word. The illustration to the right of the program shows how these scores are used by the inference engine: when the word "Glasgow" is forced after the prefix "The year begins," the candidate generation's score plummets (due to the low probability of "Glasgow"), and the generation is culled.

More generally, programs can use arbitrary logic to extend a running completion s and to multiply an *update* w into that completion's running score. These score updates can encode hard constraints (0 indicates violation), or soft constraints, based on symbolic heuristics, LM-as-judge schemes, or token probabilities under the Follower. They can also incorporate *importance weights* that correct for biases in the proposal process (see §3.3).

Mathematical interpretation of inference programs. An inference program π provides a blueprint for sequential *inference algorithms* that search for high-quality completions. Formally, we say that the goal of these algorithms is to generate samples from a *target distribution* induced by the program that assigns high probability to high-scoring sequences. Let $\sigma : \mathcal{V}^* \to P\,(\mathcal{V}^* \times \mathbb{R})$ denote a step function that maps a starting string s to a distribution $\sigma(s)$ over updated strings $s' \in \mathcal{V}^*$ and multiplicative score updates $w \in \mathbb{R}$.[1] Then for a fixed maximum generation length $T \in \mathbb{N}$, the target distribution is defined as

$$P(s) \propto \mathbb{E}_{(s'_1, w_1) \sim \sigma(\varepsilon_{\text{null}}), \ldots, (s'_T, w_T) \sim \sigma(s'_{T-1})} \left[\mathbb{1}[s'_T = s] \cdot \prod_{i=1}^{T} w_i \right]. \tag{1}$$

Intuitively, this equation defines the target probability of string s to be proportional to the probability of generating s via repeated application of `step`, upweighted or downweighted according to the score accumulated while generating it.

Scalable test-time search via probabilistic inference. Our inference engine $\mathcal{I}$ implements several general-purpose Monte Carlo methods for approximately sampling from the target distribution P. All methods support *parallel scaling* based on an inference budget N. Our experiments (§4) evaluate three instantiations of DISCIPL.

Importance Sampling (IS). Importance sampling generates N full completions $S_1, \ldots, S_N$ in parallel, by initializing N empty generations and repeatedly calling `step` until all have completed. For each candidate S_i, the score updates w from each step are multiplied to obtain an overall score W_i. Finally, one candidate S^* is selected, with $\mathbb{P}(S^* = S_i) = W_i / \sum_{j=1}^{N} W_j$. When the target distribution P is well-defined, this process converges to P as $N \to \infty$.

[1] We assume that when s is an EOS-terminated string, $\sigma(s)$ is a Dirac delta distribution at $(s, 1)$.

(A) Token masking

```python
# Generates exactly 10 chars
text = "", K = 10
while len(text) <= K:
  # Sample from mask and
  # auto-update weight
  async with TokenLengthMask(
    max_chars=K-len(text)
  ):
    text += \
      await self.next_token()
```

(B) Self-hinting

```python
# Generates gift list under a budget
async def step(self):
  # Update Follower LM prompt
  await self.hint(
    f"You have ${self.budget} left!"
  )
  gift = await self.extend(
    start="-", stop="\n"
  )
  self.budget -= self.extract_price(gift)
  self.condition(self.budget >= 0)
```

(C) Self-checking

```python
# Checks if text is a valid Haiku
def check(self, text: str) -> bool:
  lines = text.strip().split()
  if len(lines) != 3:
    return False
  # Check syllable structure
  for l, s in zip(lines, [5, 7, 5]):
    if textstat.syllable_count(l) != s:
      return False
  return True
```

Figure 2: **Inference programming patterns for self-steering**, discussed in §3.3.

Sequential Monte Carlo (SMC). Like IS, SMC (Doucet et al., 2001, Alg. 2) initializes N empty generations, called *particles*. Each particle is associated with a *weight* that is initialized to 1. SMC alternates between (1) calling `step` on all particles in parallel, multiplying the returned score update into the particle weight; and (2) *resampling* to cull low-weight particles and replace them with copies of high-scoring particles. (All weights are reset to uniform after resampling.) Having multiple copies of promising particles allows each to be extended independently by the next call to `step`. This has the effect of adaptively reallocating the computation budget (N) to focus on promising candidates. As in IS, at the end of the algorithm, we select one of the particles S^* to return, with probabilities proportional to the particle weights. Under mild technical conditions, SMC converges to P as $N \to \infty$.

Rejection Sampling (RS). When a program's `step` method generates an entire completion and then computes a binary score update, running $\mathcal{I}$ (with budget N) reduces to rejection sampling (generating N samples and checking whether any satisfy the constraint). We implement this pattern as DISCIPL-RS in §4.

3.3 Common Patterns

Programs in DISCIPL adhere to several common inference patterns, for which we have implemented library support and which are also illustrated in the prompt to the Planner.

Step-by-step problem decomposition. The Planner must decide how to decompose a task into a sequence of extend-and-score steps; this determines how often different candidates are compared and resampled. A common pattern is to make each step extend by a task-relevant unit (e.g., a line of a poem, a word of a sentence with word-level constraints, etc.).

Prior and proposal prompts. Imposing constraints can lead LMs to produce incoherent generations. For example, when prompted to generate a sentence using the words "dog," "throw," and "frisbee," small LMs yield semantically dubious completions like, "Two dogs are throwing frisbees at each other" (Lin et al., 2020). To promote coherency, programs can compensate for biases in the *proposal* distribution, which is aware of task-specific constraints, with scores from a *prior*, which ensures fluency. The Planner defines the prior and proposal distributions via separate prompts. Typically, the prior prompt defines more general instructions, e.g., "Write a sentence that is grammatically correct and makes sense."

Constrained generation with weight correction (Fig. 2A). In many situations, we might want the Follower to generate specific token sequences (e.g., "Glasgow"), or more generally, to adhere to formal constraints like regular expressions or grammars. The Planner can apply *token masks* that both enforce these constraints at generation time, and automatically incorporate *importance weights* that correct for the distortion in the LM's distribution resulting from the mask (Loula et al., 2025; Park et al., 2024).

Self-hinting (Fig. 2B). Since the Planner controls the Follower's proposal prompt, one powerful pattern is to dynamically update it to reflect stateful information relevant to the next generation step. We expose a special `hint()` method that injects "`Note to self: {hint}`" into the Follower's context, where the hint can include text as well as Python variables and objects. This technique functions as a generalized calculator (Cobbe et al., 2021) that can perform arbitrary symbolic computations and pass their results to the Follower.

Self-checking (Fig. 2C). While programs often ensure correctness by construction, some problems cannot be verified until generation is complete. In other cases, it may still be preferable to use guess-and-check over constrained generation, or to catch bugs in the inference logic. For this reason, the Planner defines a distinguished `check()` method, which (like everything it generates) can make use of external libraries.

4 EXPERIMENTS

4.1 DOMAINS

Constrained generation. We evaluate DISCIPL on COLLIE-v1 (Yao et al., 2024), a constrained generation benchmark designed as a challenge dataset for LMs. Tasks in COLLIE (Table 2) are composed from a formal grammar of constraints specified at multiple levels of text granularity—here, we focus on the sentence and paragraph levels. The combinatorial nature of the grammar is what makes COLLIE tasks challenging – Yao et al. (2024) find that overall performance ranges from 16.3% (Alpaca-7B) to 50.9% (GPT-4).

Puzzles. To evaluate generalization beyond tasks that can be defined with a grammar, we construct a mini-dataset of challenging naturalistic generation tasks. PUZZLES (Table 5) consists of four task types that require models to compose structured poetry, write a grant proposal abstract subject to various constraints, generate an ingredients list that meets a monetary budget, and plan a multi-day travel itinerary.

4.2 EVALUATION METRICS

Validity. We are interested in building systems that effectively leverage test-time compute to improve their answers to hard-to-answer queries. Accordingly, we focus on expected Pass@1, which (unlike Pass@k) models a setting that does not assume access to an oracle verifier. We implement a generalized version of the standard unbiased Pass@k estimator (Brown et al., 2024; Chen et al., 2021) that uses the weights computed during inference (if available) and uniform weights for baselines (A.4).

Coherency. As highlighted in the example in Fig. 1, the introduction of constraints can impact the fluency of generated text. To assess coherency, we adopt the LLM-as-judge evaluation from Yao et al. (2024) using GPT-4o-mini with a zero-shot prompt (A.5). While coherency could also be assessed with human ratings, this approach offers a scalable, affordable, and reasonably reliable measure of the overall text quality.

4.3 EXPERIMENT SETUP

Our experiments evaluate whether DISCIPL can generate effective and efficient inference programs for solving unseen tasks in a stratified cross-validation setup. We begin by manually writing an example inference model for a single instance of each task type in COLLIE and PUZZLES. We few-shot prompt the Planner LM with the examples from the corresponding domain, holding out the model corresponding to the target task type. (For PUZZLES, we also include the examples from COLLIE.)

Planner and Follower LMs. The goal of the Planner is to generate an `InferenceModel` subclass that implements various methods, including `step()` and `check()`. We use a system prompt in the style of a `README.md` (A.10, which also serves as a tutorial for the interested reader) to instruct to the Planner how to write inference models. We instantiate the Planner with a capable LM (`gpt-4o-2024-08-06`, OpenAI, 2024a) that can attend to detailed instructions; however, the Planner does not itself perform any reasoning outside generating Python code. We instantiate the Follower with `Llama-3.2-1B-Instruct` (Meta AI, 2024).

Baselines. We benchmark DISCIPL against several baselines:

- **Planner-only:** We prompt `GPT-4o` to directly solve the task. For comparison, we also benchmark a smaller model in the same family, `GPT-4o-mini`. We sample N independent completions according to the sample budget.
- **Follower-only:** We run `Llama-3.2-1B-Instruct` N times with `temperature=1.0`.
- **Follower-only (beam search):** We run `Llama-3.2-1B-Instruct` with stochastic beam search with beam size N and `temperature=1.0`.
- **Reasoning model:** We benchmark against a representative frontier CoT reasoning model (`o1-2024-12-17`). Following OpenAI's guidelines, we query `o1` with `max_tokens=25000`. Due

to cost, and because we expect this method to be near ceiling performance, we only query o1 once per task instance.

Expert programs. We evaluate a variant where we grant the Planner access to the reference implementation corresponding to the target task. This oracle condition (denoted DISCIPL*) evaluates the extent to which the Planner is able to recover the expert programs.

5 RESULTS AND ANALYSIS

We report the main results of our experiments in Table 1 with figures breaking down sentence (Fig. 3), paragraph (Fig. 7), and PUZZLES (Fig. 8) performance by task.

5.1 CONSTRAINT SATISFACTION

The follower-only baseline is an unreliable instruction follower. Perhaps unsurprisingly given its size, Llama-1B is only weakly able to follow the task instructions, scoring poorly on COLLIE sentence tasks (Pass@1=0.04) and PUZZLES (Pass@1=0.08) and moderately on COLLIE paragraph (Pass@1=0.60) tasks. Notably, scaling inference compute via off-the-shelf decoding methods (beam search) does not improve Llama's performance. We hypothesize that this is because beam search optimizes for sequence log probability, which is not well-correlated with the constrained generation objective.

Planner-only baseline performance is task-dependent. Compared to Llama, GPT-4o and GPT-4o-mini are more responsive to the task instructions and significantly outperform the Follower-only baselines. Nevertheless, their ability to adhere to the constraints is task-dependent: while they are nearly perfect at including specific words (e.g., `sent_04`), they struggle to position keywords at locations other than the start of a sentence (e.g., `sent_02`, `para_05`) and cannot reliably count characters (e.g., `sent_01`, `sent_03`). These results are in line with Yao et al. (2024), who found that GPT-4 performed poorly on these tasks.

Reasoning model shows strong (though not perfect) performance. With the benefit of ample test-time compute, o1 achieves near-ceiling Pass@1 on COLLIE tasks. Nevertheless, Pass@1 for o1 dips below 1.0 on several tasks (`sent_01`, `para_03`, `para_05`, and `ingredients_list`).

DISCIPL significantly boosts performance across domains. On all tasks, DISCIPL-SMC and DISCIPL-IS far exceed the Follower-only baselines, enabling various skills like character counting and word positioning that are entirely absent from Llama-3.2-1B. On paragraph tasks, DISCIPL closes most of the gap between the Follower and Planner performance, bringing Llama up to GPT-4o/GPT-4o-mini level (Fig. 7). Meanwhile, on sentence tasks, DISCIPL surpasses both Follower- and Planner-only baselines and approaches o1-level performance (Fig. 3). Finally, on PUZZLES, DISCIPL on average outperforms both Planner and Follower baselines, though performance varies between tasks and there is a bigger gap between autogenerated and expert programs.

DISCIPL knows when it is wrong. Across all tasks, autogenerated `check()` methods closely match ground truth verifiers (Fig. 5). However, rejection sampling (RS) produces substantially fewer valid generations than combining `step()` and `check()` (SMC and IS).

			COLLIE Sentence		COLLIE Paragraph		PUZZLES	
Method	Sampling Method	Model	Pass@1	Coherency	Pass@1	Coherency	Pass@1	Coherency
DisCIPL	SMC	Llama-3.2-1B	0.81	5.71	0.88	7.45	0.42	6.38
	Importance (IS)	Llama-3.2-1B	0.87	5.30	0.80	6.25	0.38	5.65
	Rejection (RS)	Llama-3.2-1B	0.11	8.61	0.79	8.21	0.25	9.02
Follower-only	Standard	Llama-3.2-1B	0.04	9.00	0.60	7.78	0.08	8.72
	Beam Search	Llama-3.2-1B	0.04	9.32	0.62	8.28	0.03	8.43
Planner-only	Standard	GPT-4o-mini	0.35	9.30	0.84	9.18	0.27	9.20
		GPT-4o	0.24	9.08	0.87	9.27	0.25	9.40
Reasoning	Standard	o1	0.95	7.66	0.99	8.82	0.82	9.00

Table 1: **Summary of results from all experiments.** Pass@1 measures expected validity. Coherency (10-point scale) measures overall fluency for all generations (including invalid ones).

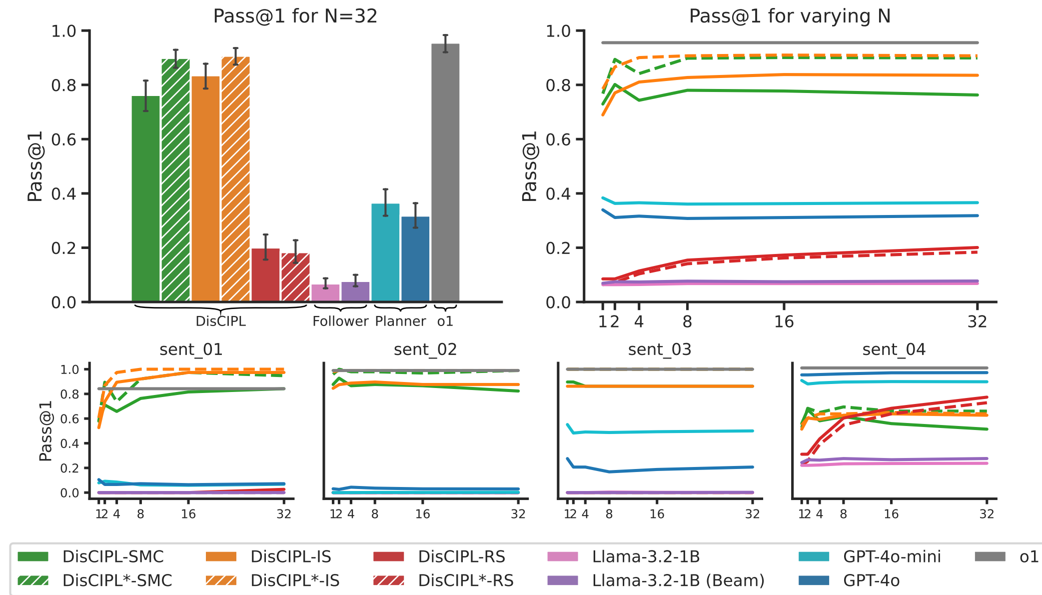

Figure 3: **Validity on COLLIE Sentence-Level Tasks.** (*Top left*) Average pass rate (Pass@1) with a fixed inference budget (N=32) corresponding to the number of samples or particles. (*Top right*) Pass@1 under a varying sample budget. While pass rates vary by task (*bottom row*), overall DISCIPL surpasses *both* the Follower and Planner baselines and approaches the performance a strong reasoning model (o1). Moreover, the performance of autogenerated inference programs nearly matches that of expert programs (DISCIPL*).

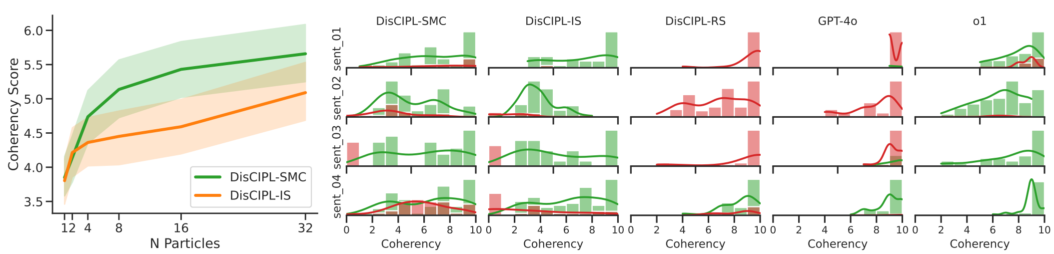

Figure 4: **Coherency.** (*Left*) SMC resampling leads to more coherent generations and scales with compute budget. (*Right*) Distributions of coherency scores on COLLIE sentence tasks (densities are normalized per-subplot). While non-reasoning LMs produce coherent but invalid text (red), DISCIPL enables Llama to satisfy constraints (green) with improving coherency as the particle count scales.

5.2 COHERENCY

What inference algorithms can we leverage to push the Pareto frontier of coherency/validity? Our results suggest that SMC (Fig. 4) is well-suited to this goal—in our experiments, SMC consistently achieves higher coherency scores than IS for comparable Pass@1. One way of understanding this result is through Fig. 1, which illustrates how SMC resampling filters out particles that satisfy constraints but introduce disfluencies. This also helps to explain why on some tasks, the Pass@1 curves for DISCIPL-SMC appear relatively flat: in these cases, the inference programs ensure validity *by construction*, so the benefits of scaling show up instead in the coherency scores.

While using a separate proposal and prior (§3.3) generally helps to ensure coherency, in some cases, it may backfire: for instance, on `sent_04` (Fig. 3) we observe that Pass@1 *decreases* with $N > 8$. This is because under the prior, the target words have low probability (they are not treated as special), so naively resampling after each word will filter out particles containing target words. Although this issue is easily avoided by choosing a step granularity that better aligns with the constraints (such as sampling multiple words until a target is generated), in practice the Planner LM has a strong inductive bias towards a word-level step, even when provided an example of an expert program.

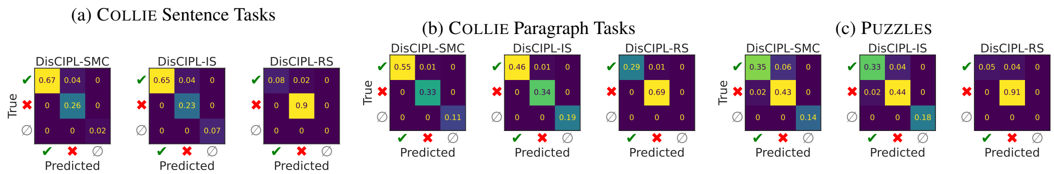

Figure 5: **Classification accuracy of autogenerated `check()` methods.** Across domains, DISCIPL aligns closely with the ground truth verifiers. (∅ indicates null results due to program errors.)

6 LIMITATIONS, FUTURE WORK, AND CONCLUSIONS

The results presented here are an early exploration of the broad framework outlined in §3. In this section, we acknowledge several limitations of the current instantiation of DISCIPL and highlight several promising directions for follow-up work.

Generalization In addition to other constrained generation settings (e.g., Lin et al., 2020; Zhou et al., 2023a), self-steering can also be extended to mathematical reasoning (Lightman et al., 2023; Wang et al., 2024a) as well as domains with soft constraints (e.g., steering based on reward models).

Inference algorithms While many text generation tasks are amenable to sequential inference, some problems may be more efficiently solved with backtracking (e.g., MCTS; Coulom, 2007; Kocsis & Szepesvári, 2006) or iterative editing (Welleck et al., 2022), both of which are implementable as extensions to DISCIPL.

Self-improvement Since generating inference programs requires non-trivial reasoning, we instantiate the Planner with a larger and more capable LM than the Follower. However, in principle, we could use the same LM to play these two roles. This recursive "self-steering" setup could learn by bootstrapping (e.g., Zelikman et al., 2022) or library learning (Ellis et al., 2021; Grand et al., 2024).

In conclusion, in this work, we introduced DISCIPL, a general framework for problem-solving with language models that orchestrates inference-time compute by writing and executing inference programs. We believe our approach offers a unifying probabilistic perspective on inference-time computation with LMs, as well as a practical tools for automating inference engineering. Our results demonstrate the potential of self-steering to enable accurate and efficient parallel inference with populations of small LMs, with performance rivaling much larger and more costly frontier models. In future work, we aim to generalize self-steering language models to new problem domains and inference patterns.

Acknowledgments

We would like to thank Timothy O'Donnell, João Loula, Ced Zhang, Cédric Colas, Noah Goodman, Aniruddha Nrusimha, and Linlu Qiu for helpful discussions and feedback. Special thanks to Ben LeBrun and the GenLM team for engineering support.

The authors gratefully acknowledge support from the MIT Quest for Intelligence, the MIT-IBM Watson AI Lab, the Intel Corporation, AFOSR, DARPA, ONR, and the National Science Foundation (NSF) under grants CCF-2217064 and IIS-2238240. G.G. is supported by a NSF Graduate Research Fellowship under Grant No. 2141064. J.B.T. is supported by AFOSR, the MIT Quest for Intelligence, the MIT-IBM Watson AI Lab, ONR Science of AI, and Siegel Family Endowment. V.K.M. and A.K.L. are supported by an anonymous philanthropic gift as well as gifts from Siegel Family Foundation that support the MIT Quest for Intelligence. J.A. is supported by a Sloan Research Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of sponsors.

Author Contributions

Gabriel Grand (primary author): Research conception, narrative development, software design and implementation, experiments, analysis of results, figure-making, writing.

Joshua B. Tenenbaum: Senior mentorship, narrative development.

Vikash K. Mansinghka: Senior mentorship, narrative development.

Alexander K. Lew: Senior mentorship, research conception, narrative development, mathematical formalisms, analysis of results, figure-making, writing.

Jacob Andreas: Senior mentorship, research conception, narrative development, analysis of results, writing.

REFERENCES

Afra Amini, Li Du, and Ryan Cotterell. Structured Voronoi Sampling, June 2024. URL http://arxiv.org/abs/2306.03061. arXiv:2306.03061 [cs].

Thomas Ball, Shuo Chen, and Cormac Herley. Can We Count on LLMs? The Fixed-Effect Fallacy and Claims of GPT-4 Capabilities, September 2024. URL http://arxiv.org/abs/2409.07638. arXiv:2409.07638 [cs].

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large Language Monkeys: Scaling Inference Compute with Repeated Sampling, December 2024. URL http://arxiv.org/abs/2407.21787. arXiv:2407.21787 [cs].

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, July 2021. URL http://arxiv.org/abs/2107.03374. arXiv:2107.03374 [cs].

Yew Ken Chia, Pengfei Hong, Lidong Bing, and Soujanya Poria. INSTRUCTEVAL: Towards Holistic Evaluation of Instruction-Tuned Large Language Models, June 2023. URL http://arxiv.org/abs/2306.04757. arXiv:2306.04757 [cs].

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training Verifiers to Solve Math Word Problems, November 2021. URL http://arxiv.org/abs/2110.14168. arXiv:2110.14168 [cs].

Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers (eds.), *Computers and Games*, pp. 72–83, Berlin, Heidelberg, 2007. Springer. ISBN 978-3-540-75538-8. doi: 10.1007/978-3-540-75538-8_7.

Mehul Damani, Idan Shenfeld, Andi Peng, Andreea Bobu, and Jacob Andreas. Learning How Hard to Think: Input-Adaptive Allocation of LM Computation, October 2024. URL https://arxiv.org/abs/2410.04707v1.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha

Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning, January 2025. URL http://arxiv.org/abs/2501.12948. arXiv:2501.12948 [cs].

David Dohan, Winnie Xu, Aitor Lewkowycz, Jacob Austin, David Bieber, Raphael Gontijo Lopes, Yuhuai Wu, Henryk Michalewski, Rif A. Saurous, Jascha Sohl-dickstein, Kevin Murphy, and Charles Sutton. Language Model Cascades, July 2022. URL http://arxiv.org/abs/2207.10342. arXiv:2207.10342 [cs].

Arnaud Doucet, Nando de Freitas, and Neil Gordon. *Sequential Monte Carlo Methods in Practice*. Springer Science & Business Media, June 2001. ISBN 978-0-387-95146-1.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 835–850, Virtual Canada, June 2021. ACM. ISBN 978-1-4503-8391-2. doi: 10.1145/3453483.3454080. URL https://dl.acm.org/doi/10.1145/3453483.3454080.

Shengyu Feng, Xiang Kong, Shuang Ma, Aonan Zhang, Dong Yin, Chong Wang, Ruoming Pang, and Yiming Yang. Step-by-Step Reasoning for Math Problems via Twisted Sequential Monte Carlo, March 2025. URL http://arxiv.org/abs/2410.01920. arXiv:2410.01920 [cs].

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-Referential Self-Improvement Via Prompt Evolution, September 2023. URL http://arxiv.org/abs/2309.16797. arXiv:2309.16797 [cs].

Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D. Goodman. Stream of Search (SoS): Learning to Search in Language, April 2024. URL http://arxiv.org/abs/2404.03683. arXiv:2404.03683 [cs].

Noah Goodman, Vikash Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models, July 2014. URL http://arxiv.org/abs/1206.3255. arXiv:1206.3255 [cs].

Gabriel Grand, Lionel Wong, Maddy Bowers, Theo X. Olausson, Muxin Liu, Joshua B. Tenenbaum, and Jacob Andreas. LILO: Learning Interpretable Libraries by Compressing and Documenting Code, March 2024. URL http://arxiv.org/abs/2310.19791. arXiv:2310.19791 [cs].

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring Mathematical Problem Solving With the MATH Dataset, November 2021. URL http://arxiv.org/abs/2103.03874. arXiv:2103.03874 [cs].

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, June 2012. ISBN 978-0-12-397795-3. Google-Books-ID: vfvPrSz7R7QC.

Chris Hokamp and Qun Liu. Lexically Constrained Decoding for Sequence Generation Using Grid Beam Search. In Regina Barzilay and Min-Yen Kan (eds.), *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1535–1546, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1141. URL https://aclanthology.org/P17-1141/.

Or Honovich, Uri Shaham, Samuel R. Bowman, and Omer Levy. Instruction Induction: From Few Examples to Natural Language Task Descriptions, May 2022. URL http://arxiv.org/abs/2205.10782. arXiv:2205.10782 [cs].

Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems, August 2024. URL http://arxiv.org/abs/2408.08435. arXiv:2408.08435 [cs].

Yuxin Jiang, Yufei Wang, Xingshan Zeng, Wanjun Zhong, Liangyou Li, Fei Mi, Lifeng Shang, Xin Jiang, Qun Liu, and Wei Wang. FollowBench: A Multi-level Fine-grained Constraints Following Benchmark for Large Language Models, October 2023. URL https://arxiv.org/abs/2310.20410v3.

Andrej Karpathy. Jagged Intelligence The word I came up with to describe the (strange, unintuitive) fact that state of the art LLMs can both perform extremely impressive tasks (e.g. solve complex math problems) while simultaneously struggle with some very dumb problems. E.g. example from two https://t.co/3C7pCdBShQ, July 2024. URL https://x.com/karpathy/status/1816531576228053133.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines, October 2023. URL http://arxiv.org/abs/2310.03714. arXiv:2310.03714 [cs].

Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou (eds.), *Machine Learning: ECML 2006*, pp. 282–293, Berlin, Heidelberg, 2006. Springer. ISBN 978-3-540-46056-5. doi: 10.1007/11871842_29.

Terry Koo, Frederick Liu, and Luheng He. Automata-based constraints for language model decoding. 2024.

Sachin Kumar, Biswajit Paria, and Yulia Tsvetkov. Gradient-based Constrained Sampling from Language Models. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 2251–2277, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.144. URL https://aclanthology.org/2022.emnlp-main.144/.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention, September 2023. URL http://arxiv.org/abs/2309.06180. arXiv:2309.06180 [cs].

Lucas Lehnert, Sainbayar Sukhbaatar, DiJia Su, Qinqing Zheng, Paul Mcvay, Michael Rabbat, and Yuandong Tian. Beyond A*: Better Planning with Transformers via Search Dynamics Bootstrapping, April 2024. URL http://arxiv.org/abs/2402.14083. arXiv:2402.14083 [cs].

Alexander K Lew, Michael Henry Tessler, Vikash K Mansinghka, and Joshua B Tenenbaum. Leveraging Unstructured Statistical Knowledge in a Probabilistic Language of Thought. *Proceedings of the Annual Meeting of the Cognitive Science Society*, pp. 7, 2020.

Alexander K. Lew, Tan Zhi-Xuan, Gabriel Grand, and Vikash K. Mansinghka. Sequential Monte Carlo Steering of Large Language Models using Probabilistic Programs, June 2023. URL http://arxiv.org/abs/2306.03081. arXiv:2306.03081 [cs, stat].

Michael Y. Li, Emily B. Fox, and Noah D. Goodman. Automated Statistical Model Discovery with Language Models, June 2024. URL http://arxiv.org/abs/2402.17879. arXiv:2402.17879 [cs].

Xiang Lisa Li, John Thickstun, Ishaan Gulrajani, Percy Liang, and Tatsunori B. Hashimoto. Diffusion-LM Improves Controllable Text Generation, May 2022. URL http://arxiv.org/abs/2205.14217. arXiv:2205.14217 [cs].

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's Verify Step by Step. October 2023. URL https://openreview.net/forum?id=v8L0pN6EOi.

Bill Yuchen Lin, Wangchunshu Zhou, Ming Shen, Pei Zhou, Chandra Bhagavatula, Yejin Choi, and Xiang Ren. CommonGen: A Constrained Text Generation Challenge for Generative Commonsense Reasoning, November 2020. URL http://arxiv.org/abs/1911.03705. arXiv:1911.03705 [cs].

Jiacheng Liu, Andrew Cohen, Ramakanth Pasunuru, Yejin Choi, Hannaneh Hajishirzi, and Asli Celikyilmaz. Don't throw away your value model! Generating more preferable text with Value-Guided Monte-Carlo Tree Search decoding, April 2024. URL http://arxiv.org/abs/2309.15028. arXiv:2309.15028 [cs].

João Loula, Benjamin LeBrun, Li Du, Ben Lipkin, Clemente Pasti, Gabriel Grand, Tianyu Liu, Yahya Emara, Marjorie Freedman, Jason Eisner, Ryan Cotterell, Vikash Mansinghka, Alexander K. Lew, Tim Vieira, and Timothy J. O'Donnell. Syntactic and semantic control of large language models via sequential monte carlo. In *The thirteenth international conference on learning representations*, 2025. URL https://openreview.net/forum?id=xoXn62FzD0.

Ximing Lu, Peter West, Rowan Zellers, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. NeuroLogic Decoding: (Un)supervised Neural Text Generation with Predicate Logic Constraints. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (eds.), *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4288–4299, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.339. URL https://aclanthology.org/2021.naacl-main.339/.

Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, Noah A. Smith, and Yejin Choi. NeuroLogic A*esque Decoding: Constrained Text Generation with Lookahead Heuristics. In Marine Carpuat, Marie-Catherine de Marneffe, and Ivan Vladimir Meza Ruiz (eds.), *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 780–799, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.naacl-main.57. URL https://aclanthology.org/2022.naacl-main.57/.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference, April 2014. URL http://arxiv.org/abs/1404.0099. arXiv:1404.0099 [cs].

Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. Probabilistic programming with programmable inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 603–616, Philadelphia PA USA, June 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192409. URL https://dl.acm.org/doi/10.1145/3192366.3192409.

R. Thomas McCoy, Shunyu Yao, Dan Friedman, Matthew Hardy, and Thomas L. Griffiths. Embers of Autoregression: Understanding Large Language Models Through the Problem They are Trained to Solve, September 2023. URL http://arxiv.org/abs/2309.13638. arXiv:2309.13638 [cs].

Meta AI. Llama 3.2: Revolutionizing edge AI and vision with open, customizable models, September 2024. URL https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show Your Work: Scratchpads for Intermediate Computation with Language Models, November 2021. URL http://arxiv.org/abs/2112.00114. arXiv:2112.00114 [cs].

OpenAI. Hello GPT-4o, May 2024a. URL https://openai.com/index/hello-gpt-4o/.

OpenAI. O1 System Card, December 2024b. URL https://cdn.openai.com/o1-system-card-20241205.pdf.

Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. Grammar-Aligned Decoding, November 2024. URL http://arxiv.org/abs/2405.21047. arXiv:2405.21047 [cs].

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models, January 2022. URL http://arxiv.org/abs/2201.11227. arXiv:2201.11227 [cs].

Matt Post and David Vilar. Fast Lexically Constrained Decoding with Dynamic Beam Allocation for Neural Machine Translation. In Marilyn Walker, Heng Ji, and Amanda Stent (eds.), *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 1314–1324, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1119. URL https://aclanthology.org/N18-1119/.

Isha Puri, Shivchander Sudalairaj, Guangxuan Xu, Kai Xu, and Akash Srivastava. A Probabilistic Inference Approach to Inference-Time Scaling of LLMs using Particle-Based Monte Carlo Methods, February 2025. URL http://arxiv.org/abs/2502.01618. arXiv:2502.01618 [cs].

Lianhui Qin, Sean Welleck, Daniel Khashabi, and Yejin Choi. COLD Decoding: Energy-based Constrained Text Generation with Langevin Dynamics, October 2022. URL http://arxiv.org/abs/2202.11705. arXiv:2202.11705 [cs].

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A Graduate-Level Google-Proof Q&A Benchmark, November 2023. URL http://arxiv.org/abs/2311.12022. arXiv:2311.12022 [cs].

Kevin Roose. When A.I. Passes This Test, Look Out. *The New York Times*, January 2025. ISSN 0362-4331. URL https://www.nytimes.com/2025/01/23/technology/ai-test-humanitys-last-exam.html.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning, June 2023. URL http://arxiv.org/abs/2303.11366. arXiv:2303.11366 [cs] version: 3.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016. ISSN 1476-4687. doi: 10.1038/nature16961. URL https://www.nature.com/articles/nature16961. Publisher: Nature Publishing Group.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters, August 2024. URL http://arxiv.org/abs/2408.03314. arXiv:2408.03314 [cs].

Jiao Sun, Yufei Tian, Wangchunshu Zhou, Nan Xu, Qian Hu, Rahul Gupta, John Frederick Wieting, Nanyun Peng, and Xuezhe Ma. Evaluating Large Language Models on Controlled Generation Tasks, October 2023. URL http://arxiv.org/abs/2310.14542. arXiv:2310.14542 [cs].

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. SynCode: LLM Generation with Grammar Augmentation, November 2024. URL http://arxiv.org/abs/2403.01632. arXiv:2403.01632 [cs].

Tim Vieira, Ben LeBrun, Mario Giulianelli, Juan Luis Gastaldi, Brian DuSell, John Terilla, Timothy J. O'Donnell, and Ryan Cotterell. From Language Models over Tokens to Language Models over Characters, December 2024. URL http://arxiv.org/abs/2412.03719. arXiv:2412.03719 [cs].

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-Shepherd: Verify and Reinforce LLMs Step-by-step without Human Annotations. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9426–9439, Bangkok, Thailand, August 2024a. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.510. URL https://aclanthology.org/2024.acl-long.510/.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-Consistency Improves Chain of Thought Reasoning in Language Models, March 2023a. URL http://arxiv.org/abs/2203.11171. arXiv:2203.11171 [cs].

Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Atharva Naik, Arjun Ashok, Arut Selvan Dhanasekaran, Anjana Arunkumar, David Stap, Eshaan Pathak, Giannis Karamanolakis, Haizhi Lai, Ishan Purohit, Ishani Mondal, Jacob Anderson, Kirby Kuznia, Krima Doshi, Kuntal Kumar Pal, Maitreya Patel, Mehrad Moradshahi, Mihir Parmar, Mirali Purohit, Neeraj Varshney, Phani Rohitha Kaza, Pulkit Verma, Ravsehaj Singh Puri, Rushang Karia, Savan Doshi, Shailaja Keyur Sampat, Siddhartha Mishra, Sujan Reddy A, Sumanta Patro, Tanay Dixit, and Xudong Shen. Super-NaturalInstructions: Generalization via Declarative Instructions on 1600+ NLP Tasks. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 5085–5109, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.340. URL https://aclanthology.org/2022.emnlp-main.340.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-Instruct: Aligning Language Models with Self-Generated Instructions, May 2023b. URL http://arxiv.org/abs/2212.10560. arXiv:2212.10560 [cs] version: 2.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. MMLU-Pro: A More Robust and Challenging Multi-Task Language Understanding Benchmark, November 2024b. URL http://arxiv.org/abs/2406.01574. arXiv:2406.01574 [cs].

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models, January 2023. URL http://arxiv.org/abs/2201.11903. arXiv:2201.11903 [cs].

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. Generating Sequences by Learning to Self-Correct, October 2022. URL http://arxiv.org/abs/2211.00053. arXiv:2211.00053 [cs].

Brandon T. Willard and Rémi Louf. Efficient Guided Generation for Large Language Models, July 2023. URL http://arxiv.org/abs/2307.09702. arXiv:2307.09702 [cs].

Lionel Wong, Gabriel Grand, Alexander K. Lew, Noah D. Goodman, Vikash K. Mansinghka, Jacob Andreas, and Joshua B. Tenenbaum. From Word Models to World Models: Translating from Natural Language to the Probabilistic Language of Thought, June 2023. URL http://arxiv.org/abs/2306.12672. arXiv:2306.12672 [cs].

Nan Xu and Xuezhe Ma. LLM The Genius Paradox: A Linguistic and Math Expert's Struggle with Simple Word-based Counting Problems, October 2024. URL http://arxiv.org/abs/2410.14166. arXiv:2410.14166 [cs].

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large Language Models as Optimizers, April 2024. URL http://arxiv.org/abs/2309.03409. arXiv:2309.03409 [cs].

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models, December 2023. URL http://arxiv.org/abs/2305.10601. arXiv:2305.10601 [cs].

Shunyu Yao, Howard Chen, Austin W Hanjie, Runzhe Yang, and Karthik Narasimhan. COLLIE: Systematic Construction of Constrained Text Generation Tasks. *ICLR*, 2024.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping Reasoning With Reasoning, May 2022. URL http://arxiv.org/abs/2203.14465. arXiv:2203.14465 [cs].

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-Taught Optimizer (STOP): Recursively Self-Improving Code Generation, August 2024. URL http://arxiv.org/abs/2310.02304. arXiv:2310.02304 [cs].

Stephen Zhao, Rob Brekelmans, Alireza Makhzani, and Roger Grosse. Probabilistic Inference in Language Models via Twisted Sequential Monte Carlo, April 2024. URL http://arxiv.org/abs/2404.17546. arXiv:2404.17546 [cs, stat].

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-Following Evaluation for Large Language Models, November 2023a. URL http://arxiv.org/abs/2311.07911. arXiv:2311.07911 [cs].

Wangchunshu Zhou, Yuchen Eleanor Jiang, Ethan Wilcox, Ryan Cotterell, and Mrinmaya Sachan. Controlled Text Generation with Natural Language Instructions, June 2023b. URL http://arxiv.org/abs/2304.14293. arXiv:2304.14293 [cs].

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large Language Models Are Human-Level Prompt Engineers, March 2023c. URL http://arxiv.org/abs/2211.01910. arXiv:2211.01910 [cs].

A APPENDIX

A.1 SMC VISUALIZATION

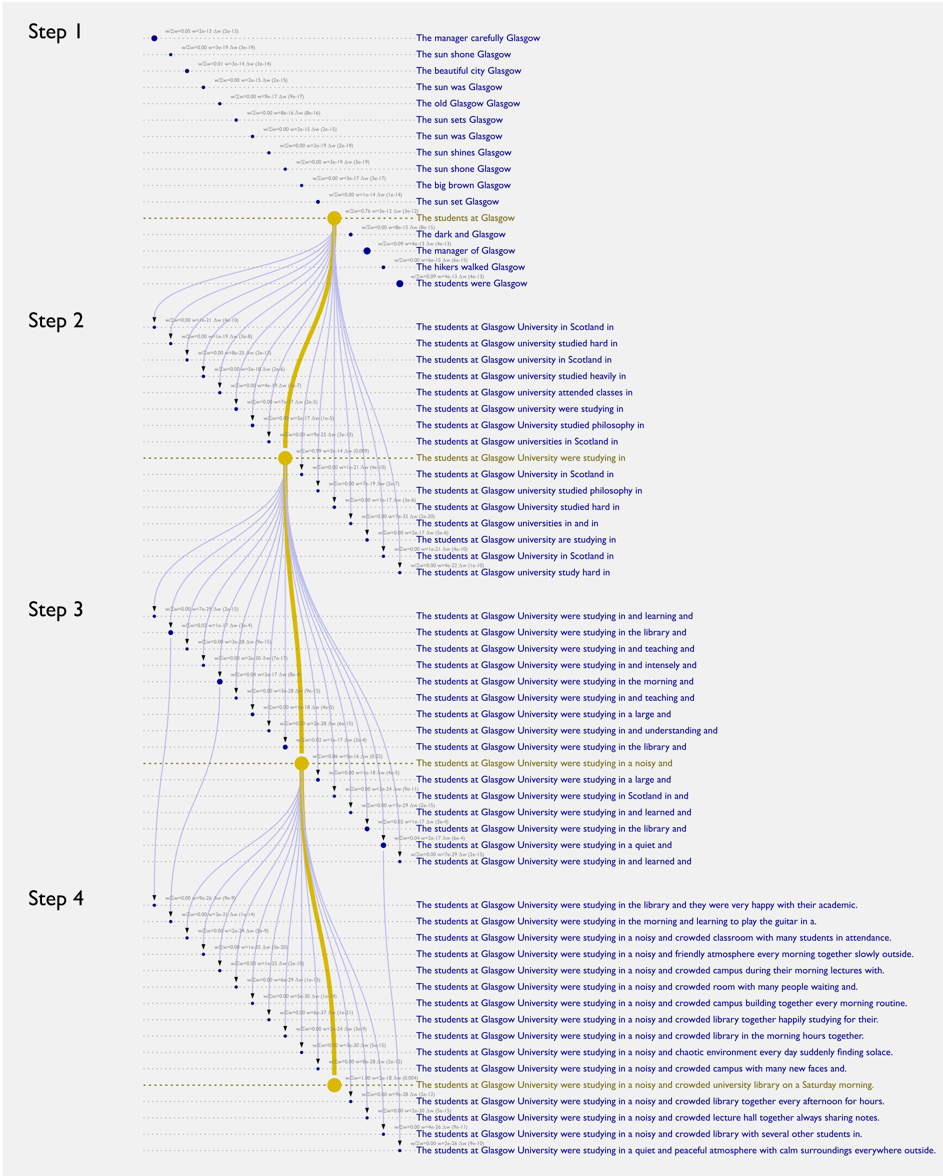

Figure 6: **Inference in action.** Visualization of SMC inference for a DISCIPL program for the COLLIE `sent_02` task (Fig. 1): "Please generate a sentence: 1) with exactly 18 words; 2) with the 4th, 8th, 11th words to be 'Glasgow', 'in', 'and' respectively." Weights for $N = 16$ particles are computed after each step and correspond intuitively to a measure of fluency under the constraints. For instance, after Step 1, particles for which "Glasgow" is a natural 4th word are propagated (e.g., "The students at Glasgow"), while others are filtered out (e.g., "The big brown Glasgow"). Each step corresponds to a single constraint: the first three steps each generate until the next target word, and the final step generates until the 18th word. In this way, the inference program ensures validity by construction, while adaptive resampling via SMC selects for overall coherency.

A.2 COLLIE Dataset and Results

We evaluate on a subset of 9/13 of the tasks in COLLIE-v1 (Yao et al., 2024) corresponding to the sentence and paragraph levels.[2] We use the task instances drawn from the Wikipedia corpus of COLLIE-v1. Since the number of instances varies significantly across task types, when computing aggregate metrics, we normalize with respect to task-level (sentence and paragraph), such that each task instance is treated as having been sampled from a uniform prior over constraint types.

Task	N	Example Prompt
sent_01	38	Please generate a sentence with exactly 82 characters. Include whitespace into your character count.
sent_02	98	Please generate a sentence: 1) with exactly 11 words; 2) with the 4th, 8th, 11th words to be ‘Series’, ‘and’, ‘4’ respectively.
sent_03	29	Please generate a sentence: 1) with at least 9 words; 2) with all words having at most 7 characters.
sent_04	94	Please generate a sentence containing the word ‘have’, ‘rising’, ‘the’.
para_01	9	Please generate a paragraph with all sentences having the 1st word to be ‘The’.
para_02	94	Please generate a paragraph: 1) with exactly 3 sentences; 2) not containing the word ‘be’; 3) not containing the word ‘this’; 4) not containing the word ‘is’.
para_03	93	Please generate a paragraph: 1) with exactly 4 sentences; 2) with all sentences having at least 12 words; 3) with all sentences having at most 20 words.
para_04	18	Please generate a paragraph: 1) with at least 3 sentences; 2) with all sentences having at least 21 words.
para_05	89	Please generate a paragraph: 1) with exactly 3 sentences; 2) with sentences having the last word to be ‘convention’, ‘president’, ‘Wisconsin’ respectively.

Table 2: **Summary of tasks in COLLIE-v1 used for our evaluation.**

		DisCIPL			DisCIPL* (expert)			Follower-only		Planner-only		Reason
		SMC	IS	RS	SMC	IS	RS	Llama-3.2-1B	+Beam	GPT-4o-mini	GPT-4o	o1
sent_01	Pass@1	0.84	0.97	0.03	0.95	1.00	0.00	0.00	0.00	0.07	0.07	0.84
	Coherency	6.71	6.95	9.39	7.55	6.58	9.39	9.24	9.71	9.66	9.16	8.03
	Error	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sent_02	Pass@1	0.81	0.87	0.00	0.98	0.98	0.00	0.00	0.00	0.00	0.03	0.98
	Coherency	4.68	3.53	6.45	5.12	4.02	6.60	6.61	7.69	8.20	7.93	6.48
	Error	0.01	0.04	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00
sent_03	Pass@1	0.86	0.86	0.00	1.00	1.00	0.00	0.00	0.00	0.50	0.21	1.00
	Coherency	4.97	4.55	9.52	7.76	5.86	9.62	9.55	9.79	9.62	9.24	7.34
	Error	0.14	0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sent_04	Pass@1	0.53	0.64	0.78	0.67	0.65	0.73	0.27	0.31	0.89	0.96	1.00
	Coherency	5.98	5.33	8.27	6.27	6.38	8.22	8.32	8.50	8.76	8.90	8.90
	Error	0.01	0.12	0.00	0.10	0.07	0.00	0.00	0.00	0.00	0.00	0.00
Overall	Pass@1	0.81	0.87	0.11	0.94	0.95	0.10	0.04	0.04	0.35	0.24	0.95
	Coherency	5.62	5.30	8.93	7.17	5.93	8.99	8.92	9.34	9.34	9.01	7.66
	Error	0.06	0.08	0.00	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00

Table 3: **COLLIE Sentence-Level Results.**

[2] While COLLIE also defines several word-level constraints, we found that, even with tools for fine-grained token masking, these present a particular challenge for LMs due to the problem of token misalignment. While it is possible to recover next-character distributions from token-level LMs (e.g., Vieira et al., 2024), such approximations are expensive to compute. Instead, we focus on constraints at the sentence-level and higher that can be effectively expressed with token-level LMs.

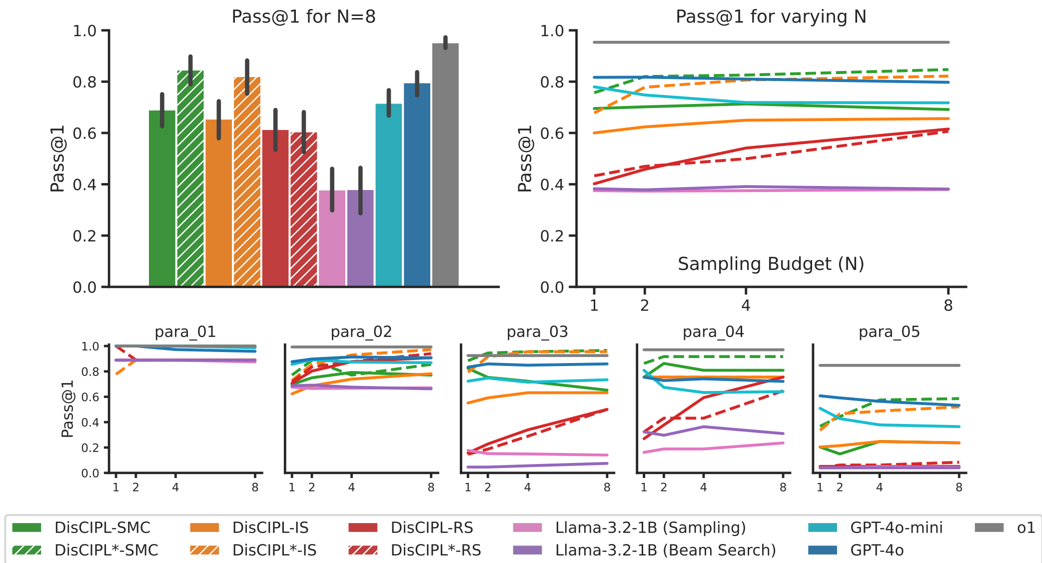

Figure 7: **Validity on COLLIE Paragraph-Level Tasks.** Figure structure is identical to COLLIE sentence-level tasks (Fig. 3). Since generations are longer, the total sampling budget is limited to $N = 8$. Baseline performance is higher for all models as these tasks appear to be more in-distribution for LMs. Nevertheless, we observe that DISCIPL helps to close the Pass@1 gap between the Follower-only (`Llama-3.2-1B`) and Planner-only (`GPT-4o`) baselines.

		DisCIPL			DisCIPL* (expert)			Follower-only		Planner-only		Reason
		SMC	IS	RS	SMC	IS	RS	Llama-3.2-1B	+Beam	GPT-4o-mini	GPT-4o	o1
para_01	Pass@1	1.00	0.89	0.89	0.89	0.89	0.89	0.88	0.89	0.99	0.96	1.00
	Coherency	7.33	5.67	7.78	8.11	8.11	8.22	7.00	7.22	9.44	9.22	8.89
	Error	0.00	0.11	0.00	0.00	0.11	0.00	0.00	0.00	0.00	0.00	0.00
para_02	Pass@1	0.78	0.79	0.91	0.86	0.98	0.95	0.68	0.67	0.88	0.91	1.00
	Coherency	7.96	7.59	9.20	9.30	9.18	9.29	9.01	9.51	9.79	9.86	8.91
	Error	0.05	0.09	0.00	0.01	0.02	0.00	0.00	0.00	0.00	0.00	0.00
para_03	Pass@1	0.65	0.62	0.48	0.98	0.97	0.48	0.10	0.03	0.73	0.87	0.94
	Coherency	5.96	5.03	8.81	8.95	7.81	9.08	9.05	9.95	9.77	9.74	9.24
	Error	0.32	0.38	0.01	0.02	0.03	0.00	0.00	0.00	0.00	0.00	0.00
para_04	Pass@1	0.83	0.78	0.78	0.94	0.78	0.67	0.24	0.32	0.66	0.74	1.00
	Coherency	8.89	8.00	9.06	8.44	7.28	9.39	8.78	9.72	9.33	9.22	9.00
	Error	0.00	0.06	0.00	0.06	0.22	0.00	0.00	0.00	0.00	0.00	0.00
para_05	Pass@1	0.20	0.20	0.01	0.56	0.49	0.04	0.00	0.00	0.33	0.51	0.83
	Coherency	3.75	3.24	7.83	4.06	3.47	7.33	7.69	9.07	7.08	8.17	6.67
	Error	0.01	0.13	0.00	0.02	0.10	0.00	0.00	0.00	0.00	0.00	0.00
Overall	Pass@1	0.88	0.80	0.79	0.89	0.84	0.76	0.60	0.62	0.84	0.87	0.99
	Coherency	7.52	6.25	8.27	8.08	7.66	8.60	7.75	8.29	9.32	9.22	8.82
	Error	0.02	0.11	0.00	0.02	0.13	0.00	0.00	0.00	0.00	0.00	0.00

Table 4: **COLLIE Paragraph-Level Results.**

A.3 PUZZLES DATASET AND RESULTS

The PUZZLES dataset consists of four task types with prompts given in Table 5. We re-run each method 10 times on each task type to produce the results in Table 6.

TASK	PROMPT		DISCIPL-SMC
Square Poem	Write a poem with 8 lines, where each line has exactly 8 words.	✓	The sun sets slowly in the evening sky Golden hues of orange and pink slowly fade The stars come out one by one tonight A night of rest for the world outside The moon is full and bright and white A beacon in the dark and endless night The world is quiet and still and deep In the stillness, all is calm and bright
Grant Proposal	Write an abstract for a grant proposal on elephant ecology and conservation. Your response should start with "Abstract: <YOUR ABSTRACT>". The abstract must be between 75-100 words. You are not allowed to use any of the following terms: conservation, sustainability, environment, ecology, wildlife, africa, asia, society, community, biodiversity, endangered, threatened, species, habitat, poaching, science, research.	✓	Abstract: Studying the complex behavior, social dynamics, and territorial boundaries of Indian elephant populations using camera trap images and GIS analysis reveals that strong elephant herds are consistently associated with increased access to food resources, particularly grasslands and aquatic habitats. These findings suggest that elephant social hierarchies play a crucial role in allocating resources and protecting against intrusions by buffalo herds. Investigating the spatial overlap between elephant and buffalo habitats, results show that elephants prioritize areas with larger buffer zones around their food sources to minimize the risk of competing utilization.
Ingredients List	Please write an ingredients list for chocolate chip brownies with at most 7 ingredients costing less than $18.00 total. The list should be in dashed bullet point format starting with "Ingredients:". Each ingredient should be listed on a separate line with the price given in USD.	✓	Ingredients: - 1 and 1/2 sticks of unsalted butter ($3.50) - 1 cup granulated sugar ($1.50) - 2 large eggs ($1.50) - 1/2 cup unsweetened cocoa powder ($1.50) - 1 teaspoon vanilla extract ($1.25) - 1 cup semi-sweet chocolate chips ($4.00)
Trip Itinerary	I'm planning a 3-day trip to Singapore. Please write me a detailed day-by-day itinerary that includes at least four activities per day. The itinerary should start with "Day 1:" and end with "Day 3:", with a blank line between each day. Each activity should be listed on a separate line starting with a time range in 24-hour format in square brackets (for example, "[11:00-13:00] Visit the Gardens by the Bay"). Make sure to leave at least 9 hours of free time each day for rest.	✓	Day 1: [06:00-07:30] Start the day at Marina Bay Sands SkyPark for panoramic views of the city. [07:30-09:30] Take a walk along the Marina Bay Waterfront Promenade for a leisurely stroll and enjoy the scenic views of the city. [09:30-12:00] Visit the Esplanade - Theatres on the Bay for a movie or a performance. [12:00-13:30] Have lunch at a nearby restaurant, such as Jumbo Seafood, and try some local seafood. [13:30-15:30] Visit the Singapore Botanic Gardens for a relaxing walk and take in the beautiful gardens. [15:30-18:00] Explore the iconic Gardens by the Bay, learn about Singapore's flora and fauna, and marvel at the stunning architecture. Day 2: [07:00-09:00] Start the day at the Singapore Zoo, home to over 2,800 animals from around the world. [09:00-11:00] Visit the nearby Universal Studios Singapore for an immersive theme park experience with rides and attractions. [11:00-12:30] Head to the iconic Merlion Park for a short walk and visit the Merlion statue. [12:30-14:30] Have lunch at a local eatery, such as Maxwell Food Centre, and try some hawker food. [14:30-16:30] Take a short taxi ride to the Gardens by the Bay for a visit to the Cloud Forest and Flower Dome. [16:30-18:00] Relax and unwind at the Gardens by the Bay with a scenic walk. Day 3: [07:00-09:00] Visit the historic Fort Canning Park, a former British military base turned public park. [09:00-11:00] Explore the nearby National Gallery Singapore, featuring a diverse collection of Southeast Asian art. [11:00-12:30] Have lunch at a nearby eatery, such as Tiong Bahru, and try some Singaporean cuisine. [12:30-14:30] Visit the nearby Little India and explore the vibrant streets of Chinatown and Little India. [14:30-16:30] Take a short taxi ride to the iconic Marina Bay Sands and visit the rooftop bar for stunning views of the city.

Table 5: **Puzzles tasks and example generations.** Samples were randomly selected from among the valid outputs produced by DISCIPL-SMC.

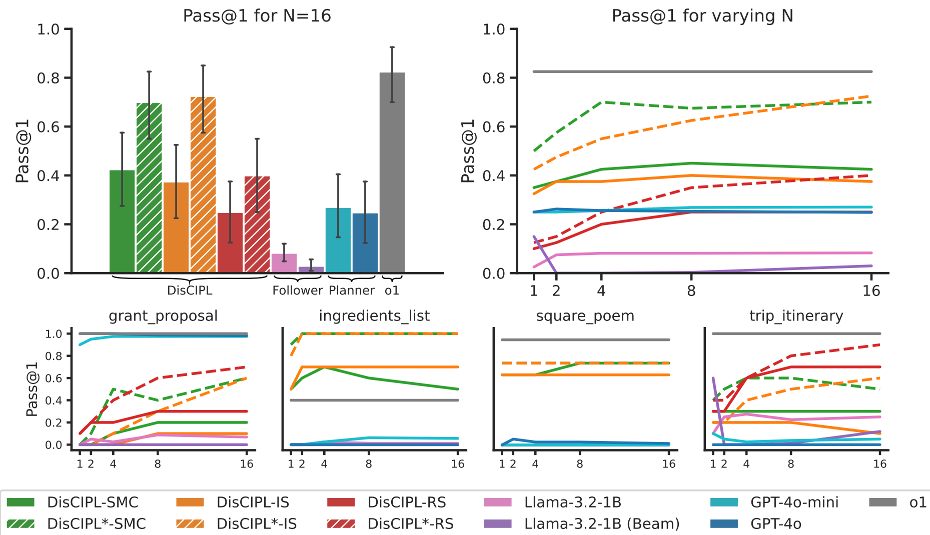

Figure 8: **Validity on PUZZLES.** Pass@1 for fixed *(top left)* and varying (top right) sample budgets for four challenge tasks *(bottom row)*. While DISCIPL still surpasses both Follower and Planner baselines, generation more often produces suboptimal programs, leading to a bigger gap vs. DISCIPL*.

		DisCIPL			DisCIPL* (expert)			Follower-only		Planner-only		Reason
		SMC	IS	RS	SMC	IS	RS	Llama-3.2-1B	+Beam	GPT-4o-mini	GPT-4o	o1
Grant Proposal	Pass@1	0.20	0.10	0.30	0.60	0.60	0.70	0.07	0.00	0.98	0.98	1.00
	Coherency	3.70	3.70	9.00	8.20	7.70	8.80	9.00	8.40	8.80	9.00	9.00
	Error	0.40	0.40	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Ingredients List	Pass@1	0.50	0.70	0.00	1.00	1.00	0.00	0.01	0.00	0.06	0.00	0.40
	Coherency	8.60	8.40	8.90	9.10	8.50	8.60	9.60	9.40	8.50	9.80	8.80
	Error	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Square Poem	Pass@1	0.70	0.60	0.00	0.70	0.70	0.00	0.00	0.00	0.00	0.01	0.90
	Coherency	8.80	6.20	9.10	7.90	6.60	9.30	9.00	7.00	9.00	9.50	8.90
	Error	0.00	0.10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Trip Itinerary	Pass@1	0.30	0.10	0.70	0.50	0.60	0.90	0.25	0.12	0.05	0.00	1.00
	Coherency	4.50	4.30	8.40	6.80	6.80	7.90	8.60	7.70	9.10	9.40	9.30
	Error	0.30	0.20	0.00	0.20	0.20	0.00	0.00	0.00	0.00	0.00	0.00
Overall	Pass@1	0.42	0.38	0.25	0.70	0.72	0.40	0.08	0.03	0.27	0.25	0.82
	Coherency	6.40	5.65	8.85	8.00	7.40	8.65	9.05	8.12	8.85	9.43	9.00
	Error	0.18	0.18	0.00	0.05	0.05	0.00	0.00	0.00	0.00	0.00	0.00

Table 6: **PUZZLES Results.**

A.4 WEIGHTED PASS@1

We formalize the the weighted Pass@1 metric used to evaluate model performance. This metric offers a natural way to incorporate sample-specific scores (e.g., particle weights w_i) into the evaluation. The approach is scale-invariant, so the absolute scale of the weights does not affect the metric.

Consider a set of N samples, indexed by $i = 1, \ldots, N$. For each sample, we define:

- A log-probability $w_i \in \mathbb{R}$. In cases where w_i is undefined (i.e., when generation produces a null output), we set its corresponding weight to zero.
- A binary pass indicator

$$I_i = \begin{cases} 1, & \text{if sample } i \text{ passes,} \\ 0, & \text{if sample } i \text{ fails.} \end{cases}$$

We define the unnormalized weight of sample i as

$$\tilde{w}_i = \exp(w_i),$$

where for methods that do not compute sample weights, we assume uniform weighting by setting $\tilde{w}_i = 1$ for all i.

Weighted Pass@1 is defined as the probability that a single sample—drawn without replacement from the N samples with probability proportional to $\tilde{w}_i$—is a passing sample. Formally, this is given by

$$\text{Weighted Pass@1} = \frac{\displaystyle\sum_{i:I_i=1} \tilde{w}_i}{\displaystyle\sum_{i=1}^{N} \tilde{w}_i}.$$

This expression represents the fraction of the total weight that corresponds to passing samples, and it reduces to the standard (unweighted) Pass@1 when all weights are equal.

A.5 COHERENCY SCORE

To assess coherency, we adopt the LLM-as-judge evaluation from Yao et al. (2024) (A.5). Whereas the original evaluation used GPT-4, to reduce inference time and cost, we use GPT-4o-mini (`gpt-4o-mini-2024-07-18`). We make minimal modifications to the original prompt to encourage GPT-4o-mini to adhere to the desired format.

```
Please concisely analyze the following text for coherency. Your analysis should be no
longer than 3 sentences and it *must* end verbatim with 'The coherency score is
<SCORE>', where <SCORE> is a number between 1 and 10.
```

A.6 Error analysis

By learning from tracebacks, DISCIPL is able to correct roughly half of all initial runtime errors (Fig. 9, left). Nevertheless, even after 3 attempts, we observer error rates of 2-6% on COLLIE (A.2) and 18% on PUZZLES (A.3). Our analysis (Fig. 9, right) finds that most errors arise from invalid token masks and timeouts. Token mask errors occur when a call to `observe()` rules out all possible next tokens (e.g., when attempting to generate punctuation when a mask that rules out punctuation is already active). Meanwhile, timeout errors are more commonly associated with infinite loops or the omission of termination conditions in the `step()` logic.

There are also cases where bugs in the generated inference programs yield incorrect outputs without triggering any errors. In A.7, we give examples of an off-by-one error (Code Block 1) and an accidental infinite loop (Code Block 2) and highlight how structural issues with the outputs can be traced back to these bugs. In future work, we aim to implement a richer form of feedback where outputs from inference are provided to the Planner LM to correct for these kinds of errors.

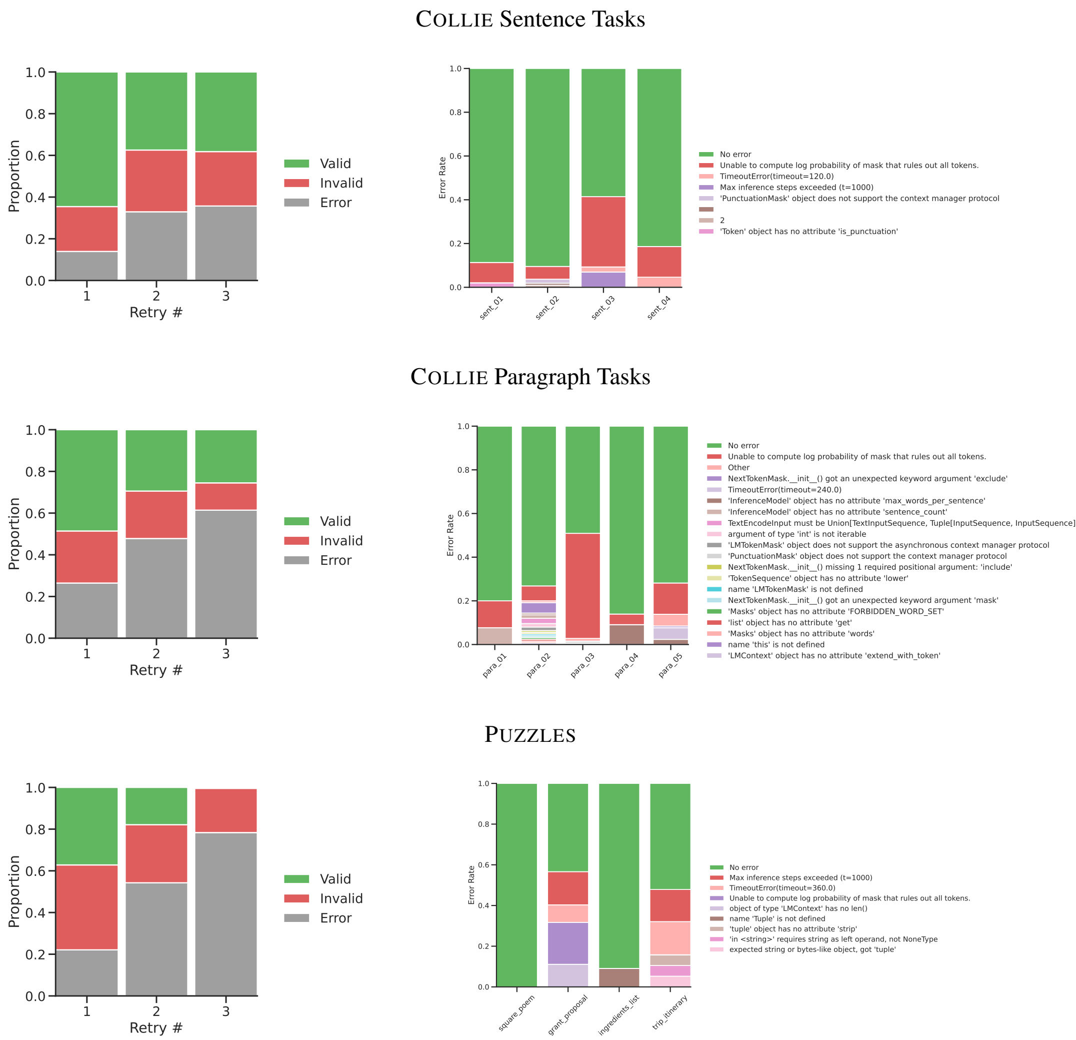

Figure 9: **Inference-time errors in DISCIPL-SMC.** (*Left*) Proportion of valid/invalid text generated at each retry, with errors in gray. Roughly half of all initial errors (#1) are corrected with feedback (#2), with diminishing returns on remaining errors for subsequent retries. (*Right*) Breakdown of Top-20 most frequent error messages aggregated across all retries.

A.7 Error Analysis: Examples

Listing 1 Example of a buggy program generated by the Planner LM for a COLLIE `sent_01` task instance. The generated program correctly applies `MaxCharsMask()` to produce a string that is exactly 102 characters. However, lines 33-34 subsequently force the Follower LM to generate punctuation, resulting in generations that exceed the target length by a single character.

```
1 class InferenceModel(BaseModel):
2     """Generates a sentence with exactly 102 characters, including whitespace."""
3
4     def __init__(
5         self,
6         context,
7         max_tokens: int = 32,
8     ):
9         super().__init__(
10             context=context,
11             max_tokens=max_tokens,
12         )
13
14         # Task-specific variables
15         self.target_length = 102
16
17     async def step(self):
18         """
19         Step granularity: token
20
21         Generation strategy: At each step, sample a token that fits within the remaining length.
22         Once the target length is reached, force punctuation and end the sentence.
23
24         End condition: Target length is reached or token limit is reached.
25         """
26
27         # Sample a token that fits within the remaining length.
28         async with MaxCharsMask(self, max_chars=self.target_length - len(self)):
29             token = await self.next_token()
30
31         # If we've reached the target length, force punctuation and end the sentence
32         if len(self) == self.target_length:
33             async with PunctuationMask(self):
34                 await self.next_token()
35             await self.end()
36             return
37
38         # Enforce token limit
39         if self.context.token_count > self.max_tokens:
40             self.condition(False)
41             await self.end()
42             return
```

Listing 2 Example of a buggy program generated by the Planner LM for a COLLIE `para_05` task instance. The `step()` function contains a while loop that samples words until the target sentence-ending word is generated. However, due to a bug in the conditional at line 34, target words containing capital letters (e.g., Oxford, College) will never be matched. As a result, the Follower LM will generate the first two sentences of the paragraph correctly and then loop indefinitely on the third sentence until it exceeds `max_tokens`.

```
1  class InferenceModel(BaseModel):
2      """Generates a paragraph with exactly 5 sentences, where each sentence ends with a specific word."""
3
4      def __init__(
5          self,
6          context,
7          max_tokens: int = 256,
8      ):
9          super().__init__(
10             context=context,
11             max_tokens=max_tokens,
12         )
13
14         # Task-specific variables
15         self.sentence_ends = ["offices", "door", "Oxford", "College", "initially"]
16         self.sentence_count = 0
17
18     async def step(self):
19         """
20         Step granularity: sentence
21
22         Generation strategy: At each step, sample a sentence until the specific last word is reached.
23         After generating the specific last word for a sentence, enforce punctuation.
24
25         End condition: All sentences are generated or token limit is reached.
26         """
27
28         # Generate a sentence until the specific last word is generated
29         target_last_word = self.sentence_ends[self.sentence_count]
30         while True:
31             word = await self.next_word()
32
33             # Check if the generated word is the target last word
34             if word.strip().lower() == target_last_word:
35                 # Force punctuation after the target last word
36                 async with PunctuationMask(self):
37                     await self.next_token()
38                 break
39
40             # Enforce token limit
41             if self.context.token_count > self.max_tokens:
42                 self.condition(False)
43                 await self.end()
44                 return
45
46         self.sentence_count += 1
47
48         # End generation after all sentences are produced
49         if self.sentence_count >= len(self.sentence_ends):
50             await self.end()
51             return
```

A.8 INFERENCE METHODS

We provide a formal definition of the SMC algorithm in DISCIPL. Our implementation is adapted from the LLAMPPL with minor adjustments intended to provide robustness to bugs in autogenerated inference programs. In particular, we set a maximum number of SMC steps $T = 1000$ so that programs that fail to terminate do not run infinitely. We also invoke this method with a wall-clock timeout (not shown) to interrupt infinite loops that might occur internally within a single `step()`.

Algorithm 2 Sequential Monte Carlo Inference Algorithm

function SMC($\pi, M_F, N, \hat{N}_{\text{ess}}, T$)
  for $i = 1, \dots, N$ **do**
    $\boldsymbol{x}^{(i)} \leftarrow \pi(M_F)$ ▷ *Instantiate particles*
  for $t = 1, \dots, T$ **do**
    await $\boldsymbol{x}$`.step()` for $\boldsymbol{x}$ in $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \dots, \boldsymbol{x}^{(N)}\}$ ▷ *Advance particle state*
    for $i = 1, \dots, N$ **do**
      $w^{(i)} \leftarrow \frac{w^{(i)}}{\sum_{j=1}^{N} w^{(j)}}$ ▷ *Normalize weights*
    if $\frac{1}{\sum_{i=1}^{N}(w^{(i)})^2} < \hat{N}_{\text{ess}}$ **then** ▷ *Compute effective sample size*
      $a^{(i)} \sim \text{Categorical}\left(\frac{w^{(1)}}{\sum_{j=1}^{N} w^{(j)}}, \dots, \frac{w^{(N)}}{\sum_{j=1}^{N} w^{(j)}}\right)$ ▷ *Resample*
      for $i = 1, \dots, N$ **do**
        $(\boldsymbol{x}^{(i)}, w^{(i)}) \leftarrow (\boldsymbol{x}^{(a^{(i)})}, \frac{1}{N}\sum_{j=1}^{N} w^{(j)})$
    if $\forall \boldsymbol{x} \in \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \dots, \boldsymbol{x}^{(N)}\} (\boldsymbol{x} \in \mathcal{V}^*_{\text{EOS}})$ **then** ▷ *Check if all particles have terminated*
      return $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \dots, \boldsymbol{x}^{(N)}\}$

A.9 IMPLEMENTATION DETAILS

LLAMPPL inference is performed with the `vLLM` backend, which uses PagedAttention (Kwon et al., 2023) for improved efficiency. For baselines, `max_tokens` is set to 32 for COLLIE sentences, 128 for COLLIE paragraphs, and 512 for PUZZLES. For DISCIPL, `max_tokens` is defined by the Planner LM and programs are executed with a max of $R = 3$ retries and a variable timeout (120s for sentences, 240s for paragraphs, and 360s for puzzles). The max sampling budget N for each domain is determined based on the number of tasks and the generation length; we use $N = 32$ for sentences, $N = 8$ for paragraphs, and $N = 16$ for puzzles.

A.10 System prompt

We use a single system prompt written in the style of a `README.md` to demonstrate to the Planner LM how to write inference models. Since this prompt is written as a condensed explanation of language model probabilistic programming, it also doubles as a useful tutorial for the interested reader. We reproduce the prompt in its entirety below.

You are an expert programmer. You are writing Python code to solve constrained generation tasks using a language model probabilistic programming language (LLaMPPL). Please read this brief tutorial on LLaMPPL and write a program to answer the user's query.

Overview

LLaMPPL is a research prototype for language model probabilistic programming: specifying language generation tasks by writing probabilistic programs that combine calls to LLMs, symbolic program logic, and probabilistic conditioning. To solve these tasks, LLaMPPL uses a specialized sequential Monte Carlo inference algorithm.

This repository (hfppl) implements LLaMPPL for use with HuggingFace Transformers.

Modeling with LLaMPPL

A LLaMPPL program is a subclass of the `hfppl.Model` class.

```
from hfppl import Model, LMContext, CachedCausalLM

# A LLaMPPL model subclasses the Model class
class InferenceModel(Model):

    # The __init__ method is used to process arguments
    # and initialize instance variables.
    def __init__(self, lm, prompt, forbidden_letter):
        super().__init__()

        # A stateful context object for the LLM, initialized with the prompt
        self.context = LMContext(lm, prompt)
        self.eos_token = lm.tokenizer.eos_token_id

        # The forbidden letter
        self.forbidden_tokens = set(i for (i, v) in enumerate(lm.vocab)
                                      if forbidden_letter in v)

    # The step method is used to perform a single 'step' of generation.
    # This might be a single token, a single phrase, or any other division.
    # Here, we generate one token at a time.
    async def step(self):
        # Condition on the next token *not* being a forbidden token.
        await self.observe(self.context.mask_dist(self.forbidden_tokens), False)

        # Sample the next token from the LLM -- automatically extends `self.context`.
        token = await self.sample(self.context.next_token())

        # Check for EOS or end of sentence
        if token.token_id == self.eos_token or str(token) in ['.', '!', '?']:
            # Finish generation
            self.finish()

    # To improve performance, a hint that `self.forbidden_tokens` is immutable
    def immutable_properties(self):
        return set(['forbidden_tokens'])
```

The Model class provides a number of useful methods for specifying a LLaMPPL program:

- `self.sample(dist[, proposal])` samples from the given distribution. Providing a proposal does not modify the task description, but can improve inference. Here, for example, we use a proposal that pre-emptively avoids the forbidden letter.
- `self.condition(cond)` conditions on the given Boolean expression.

- `self.finish()` indicates that generation is complete.
- `self.observe(dist, obs)` performs a form of ‘soft conditioning’ on the given distribution. It is equivalent to (but more efficient than) sampling a value `v` from `dist` and then immediately running `condition(v == obs)`.

To run inference, we use the `smc_standard` method:

```
import asyncio
from hfppl import smc_standard

# Initialize the HuggingFace model
lm = CachedCausalLM.from_pretrained("meta-llama/Llama-2-7b-hf")

# Create a model instance
model = InferenceModel(lm, "The weather today is expected to be", "e")

# Run inference
particles = asyncio.run(smc_standard(model, 5)) # number of particles N
```

Sample output:

```
sunny.
sunny and cool.
34° (81°F) in Chicago with winds at 5mph.
34° (81°F) in Chicago with winds at 2-9 mph.
hot and humid with a possibility of rain, which is not uncommon for this part of
Mississippi.
```

Instructions

Your goal is to implement an `InferenceModel` that encodes the user’s constraints.

The BaseModel class

To simplify writing `InferenceModel` classes, you can subclass the `BaseModel` class, which provides a number of useful methods for text generation.

```
class BaseModel(hfppl.Model):
    """Base inference model."""

    def __init__(
        self,
        context: LMContext,
        max_tokens: int = 32,
    ):
        super().__init__()
        self.context = context
        self.max_tokens = max_tokens

        self.tokenizer = context.lm.tokenizer
        self.EOS_TOKEN_ID = context.lm.tokenizer.eos_token_id

    @classmethod
    async def create(
        cls,
        lm,
        task_prompt: str = None,
        max_tokens: int = 32,
        temperature: float = 0.7,
    ):
        """Async factory method to create the model."""
        formatted_prompt = BaseModel.get_formatted_prompt(
            lm, user_prompt=cls.prior_prompt()
        )
        context = await LMContext.create(
            lm=lm,
            prompt=formatted_prompt,
            temp=temperature,
            show_eos=False,
```

```
        )
        model = cls(context, max_tokens)
        if task_prompt is not None:
            model.task_prompt = task_prompt
            formatted_task_prompt = BaseModel.get_formatted_prompt(
                lm, system_prompt=cls.system_prompt(), user_prompt=task_prompt
            )
            model.proposal_context = await LMContext.create(
                lm=lm, prompt=formatted_task_prompt, temp=temperature, show_eos=False
            )
        else:
            model.proposal_context = None
        return model

    @classmethod
    def system_prompt(cls):
        return (
            "You are helping a user generate text that satisfies constraints. "
            "Follow the user's instructions exactly. Write your response below; "
            "do not preface your response or include any additional remarks."
        )

    @classmethod
    def prior_prompt(cls):
        """A task-agnostic prompt that will be used to evaluate the prior probability of the generation."""
        raise NotImplementedError

    @staticmethod
    def get_formatted_prompt(
        lm,
        system_prompt: str = None,
        user_prompt: str = None,
        assistant_content: str = None,
    ):
        messages = BaseModel.get_chat_messages(
            system_prompt=system_prompt,
            user_prompt=user_prompt,
        )
        formatted_prompt = lm.tokenizer.apply_chat_template(
            conversation=messages,
            tokenize=False,
            add_generation_prompt=True,
        )
        if assistant_content is not None:
            formatted_prompt += assistant_content
        return formatted_prompt

    @staticmethod
    def get_chat_messages(
        system_prompt: str = None,
        user_prompt: str = None,
        system_role: str = "system",
    ):
        """Returns a list of messages in chat format."""
        messages = []
        if system_prompt is not None:
            messages += [
                {
                    "role": system_role,
                    "content": system_prompt,
                }
            ]
        if user_prompt is not None:
            messages += [{"role": "user", "content": user_prompt}]
        return messages

    def immutable_properties(self):
        return set(
            [
                "max_tokens",
                "tokenizer",
                "EOS_TOKEN_ID",
            ]
        )

    def __str__(self):
        return str(self.context)

    def __len__(self):
        return len(str(self.context))
```

```python
    async def step(self):
        """Implements a single step of the generation process.

        NOTE: This method is task-specific and should be implemented by subclasses. See the examples for guidance.
        """
        raise NotImplementedError

    async def check(self, text: str) -> bool:
        """Implements a checking procedure that will be run at the end of generation.

        Args:
            text (str): The generated text.

        Returns:
            bool: True if the text is valid, False otherwise.

        NOTE: This method is task-specific and should be implemented by subclasses. See the examples for guidance.
        """
        raise NotImplementedError

    async def sample(self, dist: hfppl.Distribution):
        return await super().sample(dist, proposal=self.get_proposal_dist(dist))

    async def observe(self, dist: hfppl.Distribution, x):
        if self.proposal_context is not None:
            await self.intervene(self.get_proposal_dist(dist), x)
        return await super().observe(dist, x)

    async def hint(self, hint_text: str):
        if self.proposal_context is None:
            return
        hint_prompt = BaseModel.get_formatted_prompt(
            lm=self.proposal_context.lm,
            system_prompt=self.system_prompt(),
            user_prompt=self.task_prompt + f"\n\n(Note to self: {hint_text})",
            assistant_content=str(self.context),
        )
        hint_context = await LMContext.create(
            lm=self.proposal_context.lm,
            prompt=hint_prompt,
            temp=self.proposal_context.temp,
            show_eos=False,
        )
        await self.intervene(hint_context.mask_dist(self.context.model_mask), True)
        self.proposal_context = hint_context

    def get_proposal_dist(self, dist: hfppl.Distribution):
        if self.proposal_context is None:
            return None
        if isinstance(dist, LMNextToken):
            return self.proposal_context.next_token()
        elif isinstance(dist, LMTokenMask):
            return self.proposal_context.mask_dist(dist.mask)
        elif isinstance(dist, NextTokenMask):
            return self.proposal_context.mask_dist(dist.mask.mask)
        else:
            raise ValueError(f"Unsupported distribution type: {type(dist)}")

    async def next_token(self) -> str:
        """Generates a single token. This method automatically extends the context with the generated token.

        Returns:
            token: hfppl.Token object representing the sampled token
        """
        # NOTE: Yields control back to the event loop. Necessary to allow timeouts to work correctly when this
        ↪ method is called in a loop.
        await asyncio.sleep(0)
        return await self.sample(self.context.next_token())

    async def next_word(
        self,
        max_chars: int = None,
    ) -> str:
        """Generates a single word. This method automatically extends the context with the generated word.

        Args:
            max_chars (int): Maximum number of characters in the word. If None, the model will sample a word of
            ↪ any length.
```

```python
        Returns:
            word: The sampled word.
        """
        await asyncio.sleep(0)

        # NOTE: This approach sometimes breaks with max_chars = 1
        if max_chars is not None:
            assert max_chars > 1

        last_token = (
            self.context.lm.str_vocab[self.context.tokens[-1]]
            if len(self.context.tokens) > 0
            else ""
        )
        last_character = last_token[-1] if len(last_token) > 0 else ""
        needs_space = (
            last_character not in string.whitespace
            and last_character
            not in [
                "-",
                "'",
                '"',
            ]
        )
        if needs_space:
            starts_word_mask = self.context.lm.masks.STARTS_NEW_WORD
        else:
            starts_word_mask = self.context.lm.masks.CONTINUES_CURRENT_WORD

        # Force model to start a new word
        await self.observe(self.context.mask_dist(starts_word_mask), True)

        word = ""
        while True:
            # Force model to sample a token with an appropriate number of characters
            if max_chars is not None:
                await self.observe(
                    self.context.mask_dist(
                        self.context.lm.masks.token_length_mask(
                            max_chars=max_chars - len(word.strip())
                        )
                    ),
                    True,
                )

            token = await self.next_token()
            word += self.context.lm.str_vocab[token.token_id]

            # If we ran out of chars, break
            if max_chars is not None and len(word.strip()) >= max_chars:
                await self.observe(
                    self.context.mask_dist(
                        self.context.lm.masks.CONTINUES_CURRENT_WORD
                    ),
                    False,
                )
                break

            # If the model wants to end the word, break
            if not (
                await self.sample(
                    self.context.mask_dist(self.context.lm.masks.CONTINUES_CURRENT_WORD)
                )
            ):
                break

        # Optionally, sample mid-word punctuation (commas, colons, hyphens, quotes, etc.)
        if await self.sample(
            self.context.mask_dist(self.context.lm.masks.MID_PUNCTUATION)
        ):
            token = await self.next_token()
            word += self.context.lm.str_vocab[token.token_id]

        return word

    def _adjust_whitespace(self, text: str) -> str:
        """Adds a prefix space to the text if it does not already start with one.
        Does not add a space if the context is empty or already ends with whitespace.
        Removes trailing whitespace from the text.
```

```
        Args:
            text (str): Text to add a prefix space to.

        Returns:
            text: Text with a prefix space added.
        """
        context = str(self.context)
        # Add a prefix space
        if (
            not text.startswith(" ")
            and not context.endswith(tuple(string.whitespace))
            and len(context) > 0
        ):
            text = " " + text

        # Ensure text does not end with whitespace, as this can cause issues with tokenization
        if text.endswith(tuple(string.whitespace)):
            print(f"Warning: Removing trailing whitespace from text: {text}")
            text = text.rstrip(string.whitespace)
        return text

    async def extend_with(self, text: str, add_prefix_space: bool = True) -> str:
        """Extends the generation with a pre-defined string literal. This method automatically extends the
        ↪  context with the input text.

        Args:
            text (str): String to extend the generation with.
            add_prefix_space (bool): Auto-add a prefix space to text. In most cases, this should be left as True.

        Returns:
            text: The generated text (same as input).
        """
        await asyncio.sleep(0)

        if add_prefix_space:
            text = self._adjust_whitespace(text)

        for token_id in self.tokenizer.encode(text, add_special_tokens=False):
            await self.observe(self.context.next_token(), token_id)

        return text

    async def extend(
        self,
        start: str = None,
        stop: List[str] = None,
        min_chars: int = None,
        max_chars: int = None,
        allow_eos: bool = True,
        add_prefix_space: bool = True,
    ) -> Tuple[str, bool]:
        """Extends the generation with a new string. This method automatically extends the context with the
        ↪  generated text.

        Args:
            start (str): String to start the generation with.
            stop (List[str]): List of strings to stop the generation at.
            min_chars (int): Minimum number of characters to generate.
            max_chars (int): Maximum number of characters to generate.
            allow_eos (bool): Allow EOS token to be generated.
            add_prefix_space (bool): Auto-add a prefix space to text. In most cases, this should be left as True.

        Returns:
            new_text: The generated text.
            eos: Whether the generation was stopped by an EOS token.
        """
        await asyncio.sleep(0)

        assert isinstance(start, (str, type(None)))
        assert isinstance(stop, (list, str, type(None)))
        assert isinstance(max_chars, (int, type(None)))
        assert isinstance(min_chars, (int, type(None)))

        if max_chars and min_chars:
            assert max_chars >= min_chars

        if isinstance(stop, str):
            stop = [stop]

        old_text = str(self.context)
```

```
        if start is not None:
            start = await self.extend_with(start, add_prefix_space=add_prefix_space)

        new_text = start or ""
        eos = False

        for _ in range(self.context.token_count, self.max_tokens):
            async with TokenLengthMask(
                self,
                max_chars=max_chars - len(new_text) if max_chars is not None else None,
                allow_eos=allow_eos and len(new_text) >= (min_chars or 0),
            ):
                token = await self.next_token()
            new_text = str(self.context)[len(old_text) :]

            # Stop on EOS token.
            if int(token) == self.EOS_TOKEN_ID:
                eos = True
                break

            # Stop on character limit.
            if max_chars is not None and len(new_text) >= max_chars:
                break

            # Stop on any stop string.
            if (
                stop is not None
                and any([s in new_text for s in stop])
                and len(new_text) >= (min_chars or 0)
            ):
                break

        return new_text, eos

    async def end(self):
        """Mark the generation as finished. This method automatically appends an EOS token if it has not already
        ↪  been generated."""
        if self.context.tokens[-1] != self.EOS_TOKEN_ID:
            async with EOSMask(self):
                await self.next_token()
        self.finish()
```

The check() method

This method implements a static Boolean checking procedure that is automatically run *once* at the end of generation to verify that the generated text satisfies the constraints. Think of `check()` as a unit test that ensures that the `InferenceModel` is working correctly. While it's useful for preventing false positives, if we only had `check()` by itself, then we would have to rely on guess-and-check. To achieve better efficiency while ensuring that outputs are correct by construction, we also need to write a sampling procedure, which is defined by the `step()` method.

The step() method

The core logic of the `InferenceModel` is the `step()` method, which is called iteratively to generate a string step-by-step via Sequential Monte Carlo sampling. The definition of a step is problem-specific – it can be a token, a word, a multi-word phrase, a line of poetry, a sentence, etc. This method can also call subroutines to invoke different types of steps at different times.

Step-by-step inference

The step function is called repeatedly as part of `smc_standard()`. Internally, the loop looks like this (with some details omitted for clarity):

```
async def smc_standard(
    model,
    n_particles: int,
    ess_threshold: float = 0.5,
```

```
):

    # Initialize the particles
    particles = [copy.deepcopy(model) for _ in range(n_particles)]

    # Keep stepping until all particles are done
    while any(map(lambda p: not p.done_stepping(), particles)):

        # Step each particle
        await asyncio.gather(*[p.step() for p in particles if not p.done_stepping()])

        # Resample according to normalized particle weights
        if ess < ess_threshold:
            ancestor_indices = [
                np.random.choice(range(len(particles)), p=weights)
                for _ in range(n_particles)
            ]
            particles = [copy.deepcopy(particles[i]) for i in ancestor_indices]

    return particles
```

Documenting design choices

There are several key design choices writing a step function that affect the accuracy and efficiency of inference. Each step() function is accompanied by a docstring that encourages the developer to consider these in their implementation.

Suppose the task is to write a sentence that includes at least three words from a list of target words. The step function docstring might look like this.

```
async def step(self):
    """
    Generation strategy:
    Each step is going to generate one of the target words.
    At each step, keep sampling words until a target word is generated.
    As soon as a target word is generated, `return` to complete the step.
    On the final step (after at least 3 target words have been generated), extend the
    sentence unconstrained until EOS.

    Step granularity: phrase

   End condition: At least 3 target words have been generated or token limit is reached.
    """
```

Step granularity

The step granularity describes how much text is generated at each call of the step function. Common step granularities include:

- **token**: each step generates a single token
- **word**: each step generates a single word
- **phrase**: each step generates multiple words
- **line**: each step generates until the newline character `"\n"`
- **sentence**: each step generates a complete sentence ending in punctuation

Because SMC resampling occurs after each step, it's important to choose the right step granularity. If the granularity is too small, each step may have a mix of particles at different points towards the solution. Since satisfying a constraint often results in lower probability under the prior, during resampling, particles that are "farther along" may unintentionally be filtered out when compared against particles that incorporate fewer constraints. On the other hand, if the granularity is too large, SMC will not be able to properly reallocate weights to more promising particles.

> [!IMPORTANT] When choosing step granularity, a rule of thumb is that a step should encapsulate a single coherent "chunk" of generation that

> satisfies a constraint or makes concrete progress towards the overall task goal.

> [!CAUTION] Avoid solving the entire problem in a single step. If your step() function contains internal loops that produce the entire generation, this is a sign that the step granularity that is too large. Instead, try breaking down the solution into multiple calls to step().

Consider the example above, where we want to write a sentence that includes at least three words from a list of target words. On the surface, this task is about words, so we might consider using a "word" step. However, most of the words that we generate will not be one of the target words, so resampling after each word is not the right granularity.

To understand why, it's important to know that the particle weights are based on the probability under a task-agnostic prior (See the section on: Prior and proposal contexts). Since the prior doesn't contain information about the target words, non-target words will have *higher* probability under the prior than target words. If we were to resample after each word, we would actually filter out generations that include the target words, which would make it difficult to satisfy the constraints.

Instead, a better strategy is for each step to keep generating words until producing a target word. Using this coarser "phrase" granularity is a good choice for this problem because it aligns the different generations in a way that allows resampling to "compare like with like." On the other hand, an even coarser granularity (e.g., generating an entire sentence at each step) is not ideal because each sentence includes multiple target words, so we lose the opportunity to resample at relevant intermediate choice points in the generation.

In general, it's simplest if the step granularity is uniform. However, in more complex problems, different steps may require different granularities: for instance, when generating an email, we might start by generating the subject line in the first step and then generate the rest of the body in subsequent steps, or as one long second step.

Generation strategy

The generation strategy section gives a high-level explanation of what generation occurs in each step and what conditions need to be met in order to ensure the constraints. For example, in the above example, the docstring implies that each step needs to sample words (i.e., using `next_word()`) and check each against the list of target words. It also suggests that there needs to be some check for when 3 target words have been generated. Rather than abruptly end as soon as this occurs, we instead want to freely generate until EOS, which is a good fit for the `extend()` method.

End condition

The end condition specifies when to end generation. There are a few common kinds of end conditions:

- All constraints were met (success)
- Some constraint was violated (failure)
- The EOS token was sampled (possibly prematurely)
- Token limit has been reached (failure)

In general, most `InferenceModel` implementations will include at least one end condition for a success state and at least one end condition for a failure (including a token limit check).

Looping and control flow

In some cases, `step()` may contain a loop that generates multiple tokens or words.

> [!CAUTION] Where possible, loops inside `step()` should be avoided in favor of a single token or word per step.

Special care should be taken to ensure that all loops are properly bounded. In general, `for` loops are preferred over `while` loops to ensure that the generation does not run indefinitely. It may be necessary to define loop bounds that are not explicitly part of the task (e.g., a max number of words in a sentence); use your best judgment.

> [!CAUTION] To ensure that `step()` yields control back to the asyncio event loop, every loop iteration should contain an `await asyncio.sleep(0)` statement. This statement is already embedded inside all methods provided by `BaseModel`, so you only need to add it manually if you are writing a custom loop that does not call any of these methods.

Masks

One of the key methods for controlling LLM generation is the concept of token masking. Token masks are simply a subset of the LLM vocabulary. In `hfppl`, there are two main patterns for interacting with masks.

Observing a mask

In many situations, it's useful to force the LLM to generate a specific token. For instance, suppose we want to generate a sentence with a fixed number of words that ends with punctuation.

In `hfppl`, this is accomplished by *observing* a mask:

```
# Get the token_ids associated with punctuation
END_PUNCTUATION = set(i for (i, v) in enumerate(lm.vocab) if v in (".", "!", "?"))

# Generates a fixed number of words
for _ in range(10):
    await self.next_word()

# Forces the LLM to generate punctuation
mask = self.context.mask_dist(END_PUNCTUATION)
await self.observe(mask, True)
punctuation_token = await self.next_token()
```

By awaiting `self.observe(mask, True)`, we guarantee that the next token sampled from the LLM will be either a period, exclamation point, or question mark.

Sampling from a mask

Sometimes, we want to whether the next token is going to be from a mask *without* forcing the LLM to generate a token from the mask. Intuitively, this is useful for checking whether the LLM "wants" to complete a generation in a certain way.

Continuing our example above, suppose want to generate a sentence, but we want the number of words to be variable. To accomplish this, we can sample from the mask:

```
# Generates a variable number of words with a max limit of 100 words
for _ in range(100):
    await self.next_word()

    # If the LLM wants to generate punctuation, then end the sentence
    if await self.sample(self.context.mask_dist(END_PUNCTUATION)):
        punctuation_token = await self.next_token()
        break
```

Mask context managers

In addition to constructing masks using `context.mask_dist()`, we also provide a higher-level interface through `NextTokenMask`. This class acts just like a token mask, but it can also be invoked as a context manager using the following syntax, which is equivalent to observing the mask:

```python
with NextTokenMask(include=END_PUNCTUATION):
    punctuation_token = await self.next_token()
```

Similarly, we can also sample from `NextTokenMask` to create conditional control flow based on whether the LLM wants to generate a particular kind of token.

```python
if await self.sample(NextTokenMask(include=END_PUNCTUATION)):
    punctuation_token = await self.next_token()
    break
```

In addition to `NextTokenMask`, we also provide several pre-defined masks for common patterns, such as punctuation, EOS, and character limits.

```python
class NextTokenMask(Distribution):
    """A context manager for masking tokens during generation.

    Provides the same interface as LMTokenMask but with the added ability to function as an async context manager.
    When used with the `with` syntax, the mask is observed, so the next token generated will be sampled from the
    ↪  mask.
    Note that the mask only applies to the next token generated, even if there are multiple tokens generated
    ↪  within the context manager.

    Args:
        model (BaseModel): The model to apply the mask to.
        include (set): The set of tokens to include in the mask. All other tokens will be excluded from
        ↪  generation.
    """

    def __init__(self, model, include: set):
        self.model = model
        self.include = include
        self.mask = LMTokenMask(model.context, include)

    def invert(self):
        """Inverts the mask so that all tokens in the mask are excluded from generation."""
        self.include = self.model.context.lm.masks.ALL_TOKENS - self.include
        self.mask = LMTokenMask(self.model.context, self.include)
        return self

    async def sample(self):
        return await self.mask.sample()

    async def log_prob(self, v):
        return await self.mask.log_prob(v)

    async def __aenter__(self):
        await self.model.observe(self.mask, True)

    async def __aexit__(self, exc_type, exc_value, traceback):
        return False

class PunctuationMask(NextTokenMask):
    """A context manager for generating end-of-sentence punctuation tokens.

    END_PUNCTUATION = [".", "!", "?"]

    Args:
        model (BaseModel): The model to apply the mask to.
    """

    def __init__(self, model):
        super().__init__(model, model.context.lm.masks.END_PUNCTUATION)

class EOSMask(NextTokenMask):
    """A context manager for generating the EOS token.

    Args:
        model (BaseModel): The model to apply the mask to.
    """

    def __init__(self, model):
        super().__init__(model, model.context.lm.masks.EOS)

class TokenLengthMask(NextTokenMask):
```

```
    """A context manager for limiting the number of characters generated.

    NOTE: Special tokens like EOS are treated as having length 0.

    Args:
        model (BaseModel): The model to apply the mask to.
        min_chars (int): The minimum number of characters to generate.
        max_chars (int): The maximum number of characters to generate.
        allow_eos (bool): Whether to allow the EOS token to be generated.
    """

    def __init__(
        self,
        model,
        min_chars: int = None,
        max_chars: int = None,
        allow_eos: bool = True,
    ):
        mask = model.context.lm.masks.token_length_mask(min_chars, max_chars)
        if len(mask) == 0:
            print(
                f"WARNING: TokenLengthMask is empty for min_chars={min_chars}, max_chars={max_chars},
                ↪  allow_eos={allow_eos}. Setting allow_eos=True."
            )

        if allow_eos or len(mask) == 0:
            mask = mask.union(model.context.lm.masks.EOS)
        else:
            mask = mask.difference(model.context.lm.masks.EOS)

        super().__init__(model, mask)

class NewLineMask(NextTokenMask):
    """A context manager for generating new line tokens.

    Args:
        model (BaseModel): The model to apply the mask to.
        n (int): The number of newlines to generate. If None, the mask will include all tokens containing a
        ↪  newline character.
    """

    def __init__(self, model, n: int = 1):
        if n is None:
            include = set(
                [i for i, v in enumerate(model.context.lm.str_vocab) if "\n" in v]
            )
        else:
            include = set([model.context.lm.str_vocab.index("\n" * n)])
        super().__init__(model, include)
```

Ending generation

Once generation is finished, we need to signal to SMC that a particle is finished, which is normally done with the `finish()` method. For convenience, `BaseModel` provides an `end()` method that ensures EOS has been generated before calling `finish()` Every `step()` implementation should contain at least one call to `end()` to ensure that it is properly terminated.

Prior and proposal contexts

During the resampling step, SMC selects for particles that have high weight under some `LMContext` distribution. In general, we want to select for generations that are grammatical and coherent. However, adhering to the task constraints can sometimes introduce disfluencies. For this reason, it is useful to separate out the *proposal distribution*, which enforces task-specific constraints, from the *prior distribution*, which encourages coherent generations. During resampling, generations are be sampled from the proposal, but scored against the prior, so that resampling selects for particles that both satisfy the constraints *and* have high probability under the prior.

For example, consider the following task:

> Please generate a sentence with exactly 11 words; with the 4th, 8th, 11th words to be 'Series', 'and', '4' respectively.

One example of a good prior prompt for this task might be, "Write a sentence that is grammatically correct and makes sense." This prompt effectively expresses the generation task while abstracting away the task-specific constraints.

The `BaseModel` class automatically implements this inference pattern by defining two separate `LMContext` distributions. During generation, these two contexts are automatically synchonized: calls to `self.sample()` and `self.observe()` update the state of both `LMContext` in tandem.

- The prior is implemented by `self.context` and uses a class-specific `self.prior_prompt` that needs to be defined for each `BaseModel`.
- The proposal is implemented by `self.proposal_context` and automatically includes the task instructions in its prompt. Additionally, the proposal context can be updated *during* generation by calling `self.hint()` as described below.

The hint() method

In many cases, it's useful to update the proposal to reflect stateful information that is relevant for the next generation step. For instance, if the task requires generating a sentence with a target character count, we could use `self.hint()` to update the proposal at each step with the number of characters remaining. In addition to tracking low-level details, hints are also useful for encouraging the model to meet higher-level constraints, such as in budgeting tasks, where the remaining budget can be recomputed symbolically inside `step()` and passed to the proposal as a hint.

Imports

You may freely import standard Python libraries when defining your `InferenceModel`. Note that the following are automatically imported in your local namespace:

```
- asyncio
- datetime
- re
- string
- nltk
- textstat
- hfppl
- BaseModel
- NextTokenMask
- PunctuationMask
- EOSMask
- TokenLengthMask
- NewLineMask
```

Examples

Next, we will take a look at some example tasks and their corresponding `InferenceModel` implementations. In these examples, the user describes the task in natural language and the assistant responds with an `InferenceModel` that encodes the user's constraints.

A.11 COLLIE EXAMPLE MODELS

```python
class CollieModelSent01(BaseModel):
    """Generates a sentence with exactly 82 characters, including whitespace."""

    def __init__(
        self,
        context,
        max_tokens: int = 32,
    ):
        super().__init__(
            context=context,
            max_tokens=max_tokens,
        )

        # Task-specific variables
        self.target_length = 82

    @classmethod
    def prior_prompt(cls):
        return "Write a sentence that is grammatically correct and makes sense."

    async def step(self):
        """
        Generation strategy:
        At each step, sample a token that fits within the remaining length.
        Once we've reached the target length, end the generation.

        Step granularity: token

        End condition: Target length is reached or token limit is reached.
        """
        remaining_length = self.target_length - len(self)

        # Provide a hint about the remaining length.
        await self.hint(f"There are {remaining_length} characters left.")

        # Sample a token that fits within the remaining length.
        async with TokenLengthMask(
            self,
            max_chars=remaining_length,
            allow_eos=(len(self) >= self.target_length),
        ):
            await self.next_token()

        # Once we've reached the maximum length, end the generation.
        # NOTE: We avoid manually sampling punctuation here as this would create an off-by-one error in the
        ↪ length constraint.
        if len(self) >= self.target_length:
            await self.end()
            return

        # Enforce token limit
        if self.context.token_count > self.max_tokens:
            await self.end()
            return

    async def check(self, text: str) -> bool:
        """Check that the generated text satisfies the length constraints."""
        return len(text) == self.target_length


class CollieModelSent02(BaseModel):
    """Generates a sentence with exactly 11 words, where the 4th, 8th, and 11th words are fixed to be 'Series',
    ↪ 'and', '4' respectively."""

    def __init__(
        self,
        context,
        max_tokens: int = 32,
    ):
        super().__init__(
            context=context,
            max_tokens=max_tokens,
        )

        # Task-specific variables
        self.word_idx = 1
        self.max_words = 11
```

```
        self.target_words = {
            4: "Series",
            8: "and",
            11: "4",
        }

    @classmethod
    def prior_prompt(cls):
        return "Write a sentence that is grammatically correct and makes sense."

    async def step(self):
        """
        Generation strategy:
        Each step is going to generate one of the target words.
        At each step, keep sampling words until a we reach a target index.
        For the target index, force the model to sample the target word.
        As soon as a target word is generated, `return` to complete the step.
        Once we hit max words, force punctuation and end.

        Step granularity: phrase

        End condition: Word index exceeds max_words or token limit is reached.
        """

        # Provide a hint about the remaining words.
        await self.hint(
            f"The following words still need to be generated: {[word for i, word in self.target_words.items() if
            ↪  i >= self.word_idx]}."
        )

        # Sample words until we hit the next target word.
        for i in range(self.word_idx, self.max_words + 1):

            # If the current word index corresponds to a target word, generate that word.
            if self.word_idx in self.target_words:
                word = self.target_words[self.word_idx]
                await self.extend_with(word)
                self.word_idx += 1

                # IMPORTANT: Return after generating the target word to complete the step.
                return

            # Otherwise, sample a word unconstrained.
            else:
                await self.next_word()
                self.word_idx += 1

        # Once max_words is reached, generate punctuation and end.
        if self.word_idx > self.max_words:
            async with PunctuationMask(self):
                await self.next_token()
            await self.end()
            return

        # Enforce token limit
        if self.context.token_count > self.max_tokens:
            self.condition(False)
            await self.end()
            return

    async def check(self, text: str) -> bool:
        """Check that the generated text satisfies the word constraints."""
        words = [w for w in nltk.word_tokenize(text) if w not in string.punctuation]
        if len(words) != self.max_words:
            return False
        for idx, word in self.target_words.items():
            if words[idx - 1].lower() != word.lower():
                return False
        return True


class CollieModelSent03(BaseModel):
    """Generates a sentence with at least 9 words, where each word has at most 7 characters."""

    def __init__(
        self,
        context,
        max_tokens: int = 32,
    ):
        super().__init__(
```

```
            context=context,
            max_tokens=max_tokens,
        )

        # Task-specific variables
        self.min_words = 9
        self.max_chars_per_word = 7

        self.word_idx = 1

    @classmethod
    def prior_prompt(cls):
        return "Write a sentence that is grammatically correct and makes sense."

    async def step(self):
        """
        Generation strategy:
        At each step, sample a word, keeping track of the word index.
        After the min_words limit is reached, allow (but do not force) end punctuation to be generated.
        Once end punctuation is generated, end the generation.
        We use max_chars to limit the length of each word.

        Step granularity: word

        End condition: Word index exceeds min_words or token limit is reached.
        """

        # Provide a hint about the number of words remaining.
        await self.hint(
            f"There are at least {self.min_words - self.word_idx} words left to generate."
        )

        # Sample a word with a maximum length of max_chars
        word = await self.next_word(
            max_chars=self.max_chars_per_word,
        )
        self.word_idx += 1

        # If we've reached the min_words limit, allow the model to end the sentence
        if self.word_idx > self.min_words:
            if await self.sample(PunctuationMask(self)):
                await self.next_token()
                await self.end()
                return

        # Enforce token limit
        if self.context.token_count > self.max_tokens:
            self.condition(False)
            await self.end()
            return

    async def check(self, text: str) -> bool:
        """Check that the generated text satisfies the word constraints."""
        words = [w for w in nltk.word_tokenize(text) if w not in string.punctuation]
        if len(words) < self.min_words:
            return False
        for word in words:
            if len(word) > self.max_chars_per_word:
                return False
        return True


class CollieModelSent04(BaseModel):
    """Generates a sentence containing the words 'have', 'rising', and 'the'."""

    def __init__(self, context, max_tokens: int = 32):
        super().__init__(
            context=context,
            max_tokens=max_tokens,
        )

        # Task-specific variables
        self.target_words = set(["have", "rising", "the"])

        # Maximum number of words before generating one of the targets
        self.max_words_per_phrase = 40

    @classmethod
    def prior_prompt(cls):
        return "Write a sentence that is grammatically correct and makes sense."
```

```python
    async def step(self):
        """
        Generation strategy:
        Each step is going to generate one of the target words.
        At each step, keep sampling words until one of the target words is generated.
        As soon as a target word is generated, `return` to complete the step.
        On the final step (once all target words have been generated), extend the sentence unconstrained until
        ↪  EOS.

        Step granularity: phrase

        End condition: All target words have been generated or token limit is reached.
        """
        # If any target words are still present, keep sampling words until one is generated.
        if len(self.target_words) > 0:

            await self.hint(
                f"The following target words are remaining: {self.target_words}."
            )

            for _ in range(1, self.max_words_per_phrase + 1):
                word = await self.next_word()

                # Remove the generated word from the target set.
                if word.strip().lower() in self.target_words:
                    self.target_words.remove(word.strip().lower())

                    # IMPORTANT: Return after generating the target word to complete the step.
                    return

                # Enforce token limit
                if self.context.token_count > self.max_tokens:
                    self.condition(False)
                    await self.end()
                    return

            # If we reach the word limit without generating a target word, reject
            self.condition(False)

        # If all target words have been generated, extend until EOS
        else:
            await self.extend()
            await self.end()

    async def check(self, text: str) -> bool:
        """Check that the generated text satisfies the word constraints."""
        words = [w for w in nltk.word_tokenize(text) if w not in string.punctuation]
        for word in self.target_words:
            if word.lower() not in words:
                return False
        return True


class CollieModelPara01(BaseModel):
    """Generates a paragraph where each sentence starts with the word 'The'."""

    def __init__(
        self,
        context,
        max_tokens: int = 128,
    ):
        super().__init__(
            context=context,
            max_tokens=max_tokens,
        )

        # Task-specific variables
        self.target_word = "The"

    @classmethod
    def prior_prompt(cls):
        return "Write a paragraph that is grammatically correct and makes sense."

    async def step(self):
        """
        Generation strategy:
        At each step, sample a sentence starting with the target word.
        Optionally end the paragraph after each sentence.
```

```python
        Step granularity: sentence

        End condition: EOS token is sampled or token limit is reached.
        """

        # Generate the sentence
        await self.extend(start=self.target_word, stop=[".", "!", "?"])

        # Allow the model to optionally end the paragraph
        if await self.sample(EOSMask(self)):
            await self.end()
            return

        # Enforce token limit
        if self.context.token_count > self.max_tokens:
            self.condition(False)
            await self.end()
            return

    async def check(self, text: str) -> bool:
        """Check that the generated text satisfies the word constraints."""
        sentences = nltk.sent_tokenize(text.lower())
        for sentence in sentences:
            if not sentence.startswith(self.target_word.lower()):
                return False
        return True


class CollieModelPara02(BaseModel):
    """Generates a paragraph with exactly 3 sentences, excluding the words 'be', 'this', and 'is'."""

    def __init__(
        self,
        context,
        max_tokens: int = 128,
    ):
        super().__init__(
            context=context,
            max_tokens=max_tokens,
        )

        # Task-specific variables
        self.n_sentences_target = 3
        self.disallowed_words = set([word.lower() for word in ["be", "this", "is"]])
        self.max_words_per_sentence = 100

        self.sentence_count = 0

    @classmethod
    def prior_prompt(cls):
        return "Write a paragraph that is grammatically correct and makes sense."

    async def step(self):
        """
        Generation strategy:
        At each step, sample a sentence word-by-word.
        If a disallowed word is generated, reject the sentence.
        Optionally end the paragraph after each sentence.

        Step granularity: sentence

        End condition: n_sentences_target is reached or token limit is reached.
        """

        end_punctuation = None

        # Provide a hint about the remaining sentences.
        await self.hint(
            f"There are {self.n_sentences_target - self.sentence_count} sentences left to generate."
        )

        # Sample the sentence word-by-word
        for _ in range(self.max_words_per_sentence):
            word = await self.next_word()

            # If the word is disallowed, reject
            if word.lower() in self.disallowed_words:
                self.condition(False)
                return
```

```
            # If we reach the end of the sentence, break
            if await self.sample(PunctuationMask(self)):
                end_punctuation = await self.next_token()
                break

        # Reject the sentence if we reach the word limit without end punctuation
        if not end_punctuation:
            self.condition(False)
            return

        self.sentence_count += 1

        # If we reach n_sentences_target, end generation
        if self.sentence_count >= self.n_sentences_target:
            await self.end()
            return

        # Enforce token limit
        if self.context.token_count > self.max_tokens:
            self.condition(False)
            await self.end()
            return

    async def check(self, text: str) -> bool:
        """Check that the generated text satisfies the word constraints."""
        sentences = nltk.sent_tokenize(text.lower())
        if len(sentences) != self.n_sentences_target:
            return False
        for sentence in sentences:
            words = [
                w for w in nltk.word_tokenize(sentence) if w not in string.punctuation
            ]
            for word in words:
                if word.lower() in self.disallowed_words:
                    return False
        return True


class CollieModelPara03(BaseModel):
    """Generates a paragraph with exactly 4 sentences, where each sentence has between 12 and 20 words."""

    def __init__(
        self,
        context,
        max_tokens: int = 128,
    ):
        super().__init__(
            context=context,
            max_tokens=max_tokens,
        )

        # Task-specific variables
        self.n_sentences_target = 4
        self.min_words_per_sentence = 12
        self.max_words_per_sentence = 20

        self.sentence_count = 0

    @classmethod
    def prior_prompt(cls):
        return "Write a paragraph that is grammatically correct and makes sense."

    async def step(self):
        """
        Generation strategy:
        At each step, sample a sentence word-by-word.
        After min_words_per_sentence have been generated, allow (but do not force) end punctuation to be
        ↪  generated.
        Once n_sentences_target is reached, end the generation.

        Step granularity: sentence

        End condition: n_sentences_target is reached or token limit is reached.
        """

        end_punctuation = None

       # Sample the sentence word-by-word, allowing end punctuation if min_words_per_sentence have been generated
        for word_idx in range(1, self.max_words_per_sentence + 1):
```

```python
            # Provide a hint about the remaining words in the sentence.
            hint_text = f"This is sentence {self.sentence_count + 1} / {self.n_sentences_target}."
            if word_idx < self.min_words_per_sentence:
                hint_text += f" There are at least {self.min_words_per_sentence - word_idx} words left to
                ↪  generate."
            else:
              hint_text += f" There are at most {self.max_words_per_sentence - word_idx} words left to generate."
            await self.hint(hint_text)

            word = await self.next_word()

            # End the sentence as soon as end punctuation is generated
            if word_idx >= self.min_words_per_sentence:
                if await self.sample(PunctuationMask(self)):
                    end_punctuation = await self.next_token()
                    break

        # Reject the sentence if we reach the word limit without end punctuation
        if not end_punctuation:
            self.condition(False)

        self.sentence_count += 1

        # If we reach n_sentences_target, end generation
        if self.sentence_count >= self.n_sentences_target:
            await self.end()
            return

        # Enforce token limit
        if self.context.token_count > self.max_tokens:
            self.condition(False)
            await self.end()
            return

    async def check(self, text: str) -> bool:
        """Check that the generated text satisfies the word constraints."""
        sentences = nltk.sent_tokenize(text.lower())
        if len(sentences) != self.n_sentences_target:
            return False
        for sentence in sentences:
            words = [
                w for w in nltk.word_tokenize(sentence) if w not in string.punctuation
            ]
            if (
                len(words) < self.min_words_per_sentence
                or len(words) > self.max_words_per_sentence
            ):
                return False
        return True


class CollieModelPara04(BaseModel):
    """Generates a paragraph with at least 3 sentences, where each sentence has at least 21 words."""

    def __init__(
        self,
        context,
        max_tokens: int = 128,
    ):
        super().__init__(
            context=context,
            max_tokens=max_tokens,
        )

        # Task-specific variables
        self.min_sentences = 3
        self.min_words_per_sentence = 21
        self.max_words_per_sentence = 100

        self.sentence_count = 0

    @classmethod
    def prior_prompt(cls):
        return "Write a paragraph that is grammatically correct and makes sense."

    async def step(self):
        """
        Generation strategy:
        At each step, sample a sentence word-by-word.
        After min_words_per_sentence have been generated, allow (but do not force) end punctuation to be
        ↪  generated.
```

```python
        After min_sentences is reached, allow (but do not force) the paragraph to end.

        Step granularity: sentence

        End condition: EOS token is sampled or token limit is reached.
        """

        end_punctuation = None

        # Sample the sentence word-by-word, allowing end punctuation if min_words_per_sentence have been generated
        for word_idx in range(1, self.max_words_per_sentence + 1):

            # Provide a hint about the remaining words in the sentence.
            hint_text = (
                f"This is sentence {self.sentence_count + 1} / {self.min_sentences}."
            )
            hint_text += f" This sentence contains {word_idx} / {self.min_words_per_sentence} words."
            await self.hint(hint_text)

            word = await self.next_word()

            # End the sentence as soon as end punctuation is generated
            if word_idx >= self.min_words_per_sentence:
                if await self.sample(PunctuationMask(self)):
                    end_punctuation = await self.next_token()
                    break

        # Reject the sentence if we reach the word limit without end punctuation
        if not end_punctuation:
            self.condition(False)
            return

        self.sentence_count += 1

        # If we reach min_sentences, optionally end the paragraph
        if self.sentence_count >= self.min_sentences:
            if await self.sample(EOSMask(self)):
                await self.end()
                return

        # Enforce token limit
        if self.context.token_count > self.max_tokens:
            await self.end()
            return

    async def check(self, text: str) -> bool:
        """Check that the generated text satisfies the word constraints."""
        sentences = nltk.sent_tokenize(text.lower())
        if len(sentences) < self.min_sentences:
            return False
        for sentence in sentences:
            words = [
                w for w in nltk.word_tokenize(sentence) if w not in string.punctuation
            ]
            if len(words) < self.min_words_per_sentence:
                return False
        return True


class CollieModelPara05(BaseModel):
    """Generates a paragraph with exactly 3 sentences, where each sentence ends with 'convention', 'president',
    ↪ 'Wisconsin'."""

    def __init__(
        self,
        context,
        max_tokens: int = 128,
    ):
        super().__init__(
            context=context,
            max_tokens=max_tokens,
        )

        # Task-specific variables
        self.n_sentences_target = 3
        self.max_words_per_sentence = 100
        self.target_last_words = [
            word.lower() for word in ["convention", "president", "Wisconsin"]
        ]
```

```
        self.sentence_count = 0

    @classmethod
    def prior_prompt(cls):
        return "Write a paragraph that is grammatically correct and makes sense."

    async def step(self):
        """
        Generation strategy:
        At each step, sample word-by-word with no punctuation.
        Once the target word is generated, add punctuation to the end of the sentence.
        Once n_sentences_target is reached, end the generation.

        Step granularity: sentence

        End condition: n_sentences_target is reached or token limit is reached.
        """

        last_word_reached = False

        # Sample the sentence word-by-word
        for word_idx in range(1, self.max_words_per_sentence + 1):

            # Provide a hint about the remaining words in the sentence.
            hint_text = f"This is sentence {self.sentence_count + 1} / {self.n_sentences_target}."
            hint_text += f" This sentence needs to end with the word
            ↪  '{self.target_last_words[self.sentence_count]}'."
            await self.hint(hint_text)

            # Sample the next word, deferring punctuation until the target word is reached
            word = await self.next_word()

            # If the word is the target word, end the sentence
            if word.strip().lower() == self.target_last_words[self.sentence_count]:
                last_word_reached = True
                break

        # Reject the sentence if we reach the word limit without the target word
        if not last_word_reached:
            self.condition(False)

        # Now add punctuation to the end of the sentence
        # Since next_word() doesn't generate punctuation, we need to do this manually
        async with PunctuationMask(self):
            await self.next_token()

        self.sentence_count += 1

        # If we reach n_sentences_target, end generation
        if self.sentence_count >= self.n_sentences_target:
            await self.end()
            return

        # Enforce token limit
        if self.context.token_count > self.max_tokens:
            self.condition(False)
            await self.end()
            return

    async def check(self, text: str) -> bool:
        """Check that the generated text satisfies the word constraints."""
        sentences = nltk.sent_tokenize(text.lower())
        if len(sentences) != self.n_sentences_target:
            return False
        for idx, sentence in enumerate(sentences):
            words = [
                w for w in nltk.word_tokenize(sentence) if w not in string.punctuation
            ]
            if words[-1].lower() != self.target_last_words[idx]:
                return False
        return True
```

A.12 PUZZLES EXAMPLE MODELS

```
class SquareWordPoemModel(BaseModel):
    """Generates a poem with N lines, where each line has exactly N words."""
```

```
    def __init__(
        self,
        context,
        max_tokens: int = 128,
    ):
        super().__init__(
            context=context,
            max_tokens=max_tokens,
        )

        # Task-specific variables
        self.N = 8  # Number of lines and words per line
        self.line_i = 1

    @classmethod
    def prior_prompt(cls):
        return "Write a poem."

    async def step(self):
        """
        Generation strategy:
        Each step is going to generate a line of the poem.
        To generate a line, sample word-by-word N times.
        After generating a line, sample a newline token to move to the next line.

        Step granularity: line

        End conditions:
        1. N lines are generated.
        2. The token limit is reached.
        """
        for _ in range(self.N):
            await self.next_word()
        async with NewLineMask(self, n=1):
            await self.next_token()
            self.line_i += 1
        if self.line_i > self.N:
            await self.end()
        # Enforce token limit
        if self.context.token_count > self.max_tokens:
            self.condition(False)
            await self.end()
            return

    async def check(self, text: str) -> bool:
        lines = text.strip().split("\n")
        if len(lines) != self.N:
            return False
        for line in lines:
            words = line.split()
            if len(words) != self.N:
                return False
        return True


class GrantProposalModel(BaseModel):
    "Generates an abstract for a grant proposal on elephant ecology and conservation. The abstract starts with
    ↪  'Abstract:'. It is between 75 and 100 words and excludes a list of forbidden words."

    def __init__(
        self,
        context,
        max_tokens: int = 512,
    ):
        super().__init__(
            context=context,
            max_tokens=max_tokens,
        )

        # Task-specific variables
        self.forbidden_words = set(
            [
                "conservation",
                "sustainability",
                "environment",
                "ecology",
                "wildlife",
                "africa",
                "asia",
```

```
                "society",
                "community",
                "biodiversity",
                "endangered",
                "threatened",
                "species",
                "habitat",
                "poaching",
                "science",
                "research",
            ]
        )
        self.min_words = 75
        self.max_words = 100
        self.word_count = 0
        self.header = False

    @classmethod
    def prior_prompt(cls):
        return "Write an abstract for a grant proposal on elephant ecology and conservation."

    async def step(self):
        """
        Generation strategy:
        Each step is going to generate a word for the abstract.
        On the first step, we generate the header, "Abstract:".
        On subsequent steps, we sample a word and check if it is a forbidden word.
        If the word is forbidden, reject.
        After each word, check if the model wants to sample punctuation.
        If the minimum word count is reached, additionally allow the model to sample EOS.
        If the maximum word count is reached, reject.

        Step granularity: word

        End conditions:
        1. The model samples EOS.
        2. The maximum word count is reached.
        3. The token limit is reached.
        """

        # Generate the title first.
        if not self.header:
            await self.extend_with("Abstract:")
            self.header = True
            return

        hint_text = f"The current length is {self.word_count} words."
        if self.word_count < self.min_words:
            hint_text += (
                f" We need at least {self.min_words - self.word_count} more words."
            )
        else:
           hint_text += f" There are only {self.max_words - self.word_count} words left before we hit the limit!"
        await self.hint(hint_text)

        # Sample a word.
        word = await self.next_word()
        self.word_count += 1

        # Check if the sentence contains any forbidden words.
        if word.strip().lower() in self.forbidden_words:
            self.condition(False)
            return

        # Optionally, sample punctuation (but do not end, since the abstract will contain multiple sentences).
        if await self.sample(PunctuationMask(self)):
            await self.next_token()

        # If the minimum word count has been reached, allow the model to sample EOS.
        if self.min_words <= self.word_count < self.max_words:
            if await self.sample(EOSMask(self)):
                await self.end()
                return
        # If the maximum word count has been reached, reject.
        elif self.word_count >= self.max_words:
            self.condition(False)
            await self.end()
            return

        # Enforce token limit.
```

```
        if self.context.token_count > self.max_tokens:
            self.condition(False)
            await self.end()
            return

    async def check(self, text: str) -> bool:
        text = text.strip()

        # Check for the header
        if not text.startswith("Abstract:"):
            return False

        # Extract abstract
        abstract = text[len("Abstract:") :].strip()
        words = abstract.lower().split()

        # Check word count
        word_count = len(words)
        if word_count < self.min_words or word_count > self.max_words:
            return False

        # Check for forbidden words first
        for word in words:
            if word in self.forbidden_words:
                return False

        return True


class IngredientsListModel(BaseModel):
    """Writes an ingredients list for chocolate chip brownies with at most 7 ingredients costing less than
    ↪ $18.00 total. The list is in bullet point format starting with "Ingredients:". Each ingredient is listed
    ↪ on a separate line with the price given in USD."""

    def __init__(
        self,
        context,
        max_tokens: int = 256,
    ):
        super().__init__(
            context=context,
            max_tokens=max_tokens,
        )

        # Task-specific variables
        self.max_ingredients = 7
        self.max_cost = 18.00

        self.line_i = 0
        self.total_cost = 0.0

    @classmethod
    def prior_prompt(cls):
        return "Write an ingredients list for chocolate chip brownies."

    def extract_cost(self, text: str) -> float:
        match = re.search(r"\$(\d+(?:\.\d+)?)", text)
        if not match:
            return None
        try:
            cost = float(match.group(1))
        except ValueError:
            return None

        cost = round(cost, 2)
        return cost

    async def step(self):
        """
        Generation strategy:
        Each step is going to generate an ingredient for the list.
        To generate an ingredient, sample the ingredient name and price.
        We can use a hint to inform the model of the remaining budget.
        If the cost of the ingredient exceeds the maximum cost, reject.
        After generating an ingredient, check if the model wants to sample EOS.
        If the maximum number of ingredients is reached, force the model to sample EOS.

        Step granularity: line

        End conditions:
```

```
        1. The model samples EOS.
        2. The maximum number of ingredients is reached.
        3. The cost limit is reached.
        4. The token limit is reached
        """
        # The first step generates the header "Ingredients:"
        if self.line_i == 0:
            await self.extend_with("Ingredients:\n")
            self.line_i += 1
            return

        # Provide a hint about the remaining budget
        await self.hint(
            f"The remaining budget is ${self.max_cost - self.total_cost:.2f}."
        )

        # Generate the ingredient
        # Ensure the line starts with a hyphen and ends with a newline
        # Set allow_eos=True to allow the model to sample EOS
        ingredient, eos = await self.extend(start="-", stop=["\n"], allow_eos=True)

        # Extract the cost of the ingredient
        cost = self.extract_cost(ingredient)
        if cost is None:
            self.condition(False)
            return

        # Update the running total cost
        self.total_cost += cost
        if self.total_cost > self.max_cost:
            self.condition(False)
            return

        # If the model sampled EOS on this step, end the generation
        if eos:
            await self.end()
            return
        # If the maximum number of ingredients is reached, force the model to sample EOS
        elif self.line_i >= self.max_ingredients:
            await self.end()
            return

        # Enforce token limit
        if self.context.token_count > self.max_tokens:
            self.condition(False)
            await self.end()
            return

        self.line_i += 1

    async def check(self, text: str) -> bool:
        lines = text.strip().split("\n")
        if lines[0] != "Ingredients:":
            return False
        if len(lines) > self.max_ingredients + 1:
            return False

        total_cost = 0
        for line in lines[1:]:
            if not line:
                continue
            if not line.startswith("-"):
                return False
            cost = self.extract_cost(line)
            if cost is None:
                return False
            total_cost += cost

        if total_cost > self.max_cost:
            return False

        return True


class TripItineraryModel(BaseModel):
    """Generates a three-day trip itinerary with at least 4 activities per day. Each day should start with "Day
    ↪  N:" and end with a double newline. Each activity should start with a time range in 24-hour format in
    ↪  square brackets (e.g., "[11:00-13:00]") and end with a newline. The itinerary should include at least 9
    ↪  hours of free time each day for rest."""
```

```python
    def __init__(
        self,
        context,
        max_tokens: int = 512,
    ):
        super().__init__(
            context=context,
            max_tokens=max_tokens,
        )

        # Task-specific variables
        self.n_days = 3
        self.n_activities = 4
        self.day_i = 1
        self.activity_i = 0

        self.min_free_time = 9

    @classmethod
    def prior_prompt(cls):
        return "Write a day-by-day itinerary for a 3-day trip to Singapore."

    def extract_time_range(
        self, text: str
    ) -> Tuple[datetime.datetime, datetime.datetime]:
        match = re.search(r"\[\d{2}:\d{2}-\d{2}:\d{2}\]", text)
        if not match:
            return None
        time_range = match.group(0)
        # Remove square brackets and split into start and end times
        time_range_clean = time_range.strip("[]")
        try:
            start_str, end_str = time_range_clean.split("-")
            start_time = datetime.datetime.strptime(start_str, "%H:%M")
            end_time = datetime.datetime.strptime(end_str, "%H:%M")
        except ValueError:
            return None
        return start_time, end_time

    def is_complete(self) -> bool:
        return self.day_i >= self.n_days and self.activity_i >= self.n_activities

    async def step(self):
        """
        Generation strategy:
        Each step is going to generate a line in the itinerary consisting of a time range and an activity.
        Each line should start with a time range in 24-hour format (e.g., "[11:00-13:00]") and end with a single
        ↪ or double newline.
        If the line ends with a double newline, check to make sure there are at least 4 activities for the day
        ↪ and move to the next day.
        After enough activities are generated on the final day, allow the model to sample EOS to end the
        ↪ itinerary.

        Step granularity: line

        End conditions:
        1. The model ends the itinerary after generating enough activities for each day.
        2. The model prematurely ends the itinerary without generating enough activities.
        3. The free time limit is reached.
        4. The token limit is reached.
        """
        # Generate the header for the day.
        if self.activity_i == 0:
            await self.extend_with(f"Day {self.day_i}:\n")
            self.activity_i += (
                1  # IMPORTANT: Increment activity_i to avoid infinite loop.
            )
            self.free_time = 24
            return

        # Generate the activity.
        # Ensure the line starts with a time range in 24-hour format and ends with a newline.
        # On the final day, once the model generates enough activities, allow it to sample EOS.
        activity, eos = await self.extend(
            start="[", stop=["\n"], allow_eos=self.is_complete()
        )

        # Extract the time range of the activity.
        time_range = self.extract_time_range(activity)
        if time_range is None:
```

```python
                self.condition(False)
                return

            # Check if there is enough free time for the day.
            start_time, end_time = time_range
            activity_duration = (end_time - start_time).seconds / 3600
            self.free_time -= activity_duration
            if self.free_time < self.min_free_time:
                self.condition(False)
                return

            # If the model sampled EOS on this step, end the generation.
            if eos:
                await self.end()
                return

            # If the model generated a double newline, move to the next day.
            if activity.endswith("\n\n"):

                # If there aren't enough activities for the day, reject.
                if self.activity_i < self.n_activities:
                    self.condition(False)
                    return

                # If this is the final day, force the model to end.
                if self.day_i == self.n_days:
                    await self.end()
                    return

                # Move to the next day.
                self.day_i += 1
                self.activity_i = 0
                return

            # Enforce token limit.
            if self.context.token_count > self.max_tokens:
                self.condition(False)
                await self.end()
                return

            # Move to the next activity.
            self.activity_i += 1

        async def check(self, text: str) -> bool:
            days = text.strip().split("\n\n")
            if len(days) != 3:
                return False

            for day in days:
                lines = day.split("\n")
                if not lines[0].startswith("Day"):
                    return False

                activities = lines[1:]
                if len(activities) < 4:
                    return False

                free_time = 24
                for activity in activities:
                    time_range = self.extract_time_range(activity)
                    if time_range is None:
                        return False
                    start_time, end_time = time_range
                    activity_duration = (end_time - start_time).seconds / 3600

                    # Ensure the activity duration is non-negative.
                    if activity_duration < 0:
                        return False
                    free_time -= activity_duration

                # Ensure there is enough free time for the day.
                if free_time < self.min_free_time:
                    return False

            return True
```