# Detecting and Characterizing Planning in Language Models

**Anonymous Author(s)**
Affiliation
Address
email

## Abstract

Modern large language models (LLMs) have demonstrated impressive performance across a wide range of multi-step reasoning tasks. Recent work suggests that LLMs may perform *planning* — selecting a future target token in advance and generating intermediate tokens that lead towards it — rather than merely *improvising* one token at a time. However, existing studies assume fixed planning horizons and often focus on single prompts or narrow domains. To distinguish planning from improvisation across models and tasks, we present formal and causally grounded criteria for detecting planning and operationalize them as a semi-automated annotation pipeline. We apply this pipeline to both base and instruction-tuned Gemma-2-2B models on the MBPP code generation benchmark and a poem generation task where Claude 3.5 Haiku was previously shown to plan. Our findings show that planning is not universal: unlike Haiku, Gemma-2-2B solves the same poem generation task through improvisation, and on MBPP it switches between planning and improvisation across similar tasks and even successive token predictions. We further show that instruction tuning refines existing planning behaviors in the base model rather than creating them from scratch. Together, these studies provide a reproducible and scalable foundation for mechanistic studies of planning in LLMs.

## 1 Introduction

Large language models (LLMs) have achieved impressive results on complex reasoning tasks, from creative writing to code generation [1, 2]. These tasks often require multi-step reasoning, yet LLMs are trained as next-token predictors that would presumably generate outputs by *improvising* each token step-by-step without foresight. An alternative hypothesis is that LLMs solve these tasks by *planning*: using intentional processes, whether internal or external, that guide generation in a structured, goal-directed way to improve coherence and reasoning. This distinction is critical because planning may be necessary for reliable chain-of-thought reasoning and long-horizon problem solving, while hidden planning mechanisms could enable models to pursue unintended goals or conceal their reasoning. Therefore, it is essential to determine not just whether or not planning occurs, but also the *mechanisms* through which planning arises.

Recent work provides evidence that LLMs engage in internal planning. For example, researchers have probed for representations of future tokens at fixed distances ahead in language models [3, 4, 5] and games like chess [6, 7] and Sokoban [8]. Lindsey et al. [9] showed that Claude 3.5 Haiku was shown to generate a poem by storing candidate rhyme words before writing the next line, and ablating the representations for these rhyme words alone can potentially change the entire next line. However, these studies have key limitations: they assume fixed planning horizons, need task-specific probes, or work only in narrow domains. Without unified and robust tools for distinguishing between planning

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

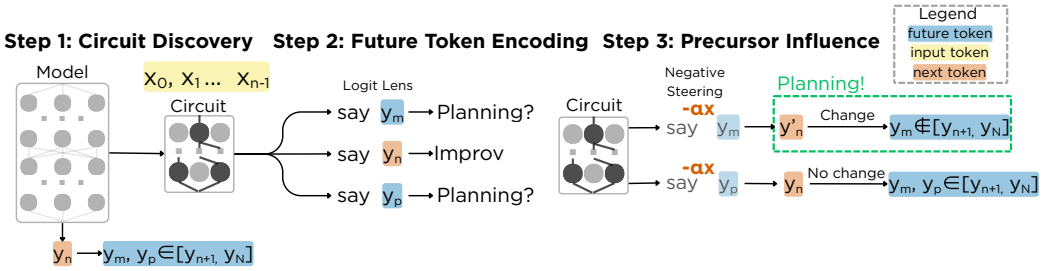


Figure 1: Planning detection at a glance. While predicting the next token $y_n$, we ask if the model is already planning for a later token $y_m$. *Step 1* isolates a smaller SAE-feature circuit that causes $y_n$. *Step 2* (Future-Token Encoding, FTE) uses a Logit-Lens readout to see which features "write" towards a future token (blue) versus the next token (red); future-writing features are planning candidates. *Step 3* (Precursor Influence, PI) negatively steers each candidate at its earliest use (orange $-\alpha x$). If this steering changes the future token $y_n$, perturbs intermediate tokens, and removes $y_m$ from the output, we call it PLANNING.

and improvisation, it is difficult to systematically compare behaviors across architectures and prompts or understand how planning in LLMs fundamentally works.

In this work, we address these gaps by formalizing a general, falsifiable definition of planning that applies to arbitrary LLMs and tasks (§2). Our approach makes two technical advances. First, we translate the intuitive notion of planning into concrete, causally grounded criteria at the activation level and provide an operational pipeline for annotating a forward pass as planning or improvisation. Second, we apply this pipeline to the planning in poems example from Lindsey et al. [9] as well as a subset of the MBPP [2] code generation benchmark. Our findings include the following:

- In contrast with Lindsey et al. [9], who found clear evidence for backward planning in Claude on a rhyming task, we find that Gemma-2-2B solves the same task successfully by improvisation, without explicit intermediate planning signals (§3.1).
- When we generalized to the larger MBPP benchmark, we found that Gemma-2-2B switches between planning and improvisation within tasks and even within successive token predictions. We also demonstrate the examples where our criteria do not have a sure answer (§3.2.4).
- In all cases where Gemma-2-2B Instruct can solve an MBPP task, Gemma-2-2B Base consistently solves the improvisation cases correctly but shows lower performance on cases where planning is involved. The base model still engages in planning, but executes it less effectively. (§3.3). We find evidence that instruction tuning refines planning behavior rather than creating it.

With these studies, alongside an explicit and reproducible pipeline for verifying planning and improvisation, we aim to scale up and advance mechanistic studies of LLM reasoning.

## 2 Defining and Detecting Planning

We aim to turn the intuitive idea of planning in LLMs into a general-purpose, and empirically verifiable definition. This section introduces the formal criteria we use throughout the paper and an operational pipeline for verifying it on real datasets and models.

### 2.1 Motivating Example

Consider the task of completing a rhyming couplet as studied by Lindsey et al. [9], where the model is given the following prompt:

*A rhyming couplet:*`\n` *He saw a carrot and had to grab it,*`\n`

In this task, the model is expected to return a line that rhymes with the end words "grab it". To write the next line, the model could use either of the following strategies:

- **Improvisation.** Generate each token one-by-one and choose a rhyming end word (e.g. “rabbit” and “habit”) only at the final position.
- **Planning.** Decide on the rhyming word in advance (e.g. “rabbit”) and generate each subsequent token to ensure the line ends with the chosen word.

Lindsey et al. [9] investigated a variant of the above prompt in Claude 3.5 Haiku [10] and demonstrated that it plans ahead for two possible rhyming words, “rabbit" and “habit". To do this, they showed that internal representations of the words “rabbit" and “habit" are active at the next position after the prompt. By default, the model returned a line ending with “like a starving rabbit", but suppressing features associated with the word “rabbit" led to the model returning “a powerful habit” instead. The fact that suppressing the “rabbit" features led the model towards a different output indicates that those “rabbit" features had a causal effect on the resulting output.

This intuition suggests that planning requires an internal representation of a future token that is active at an earlier position in the sequence and causally influences the generation of all subsequent tokens leading up to it. We formalize these two requirements below.

### 2.2 Formalizing Planning

In this paper we use sparse autoencoder (SAE) latents [11] as interpretable representations of tokens and concepts in the model. We denote these representations through triples $(l, f, t)$, where $l$ is a layer index, $f$ is an SAE latent, and $t$ is a token position. Consider a prompt $(x_0, \ldots, x_{n-1})$ and let $(y_n, \ldots, y_N)$ be the output tokens generated by the model. We call $y_n$ the *current* token and $y_m$ $(m > n)$ a *future* token.

**Definition 1** (Future-Token Encoding)**.** Let $W_l[f]$ be the decoding direction for latent $f$ at layer $l$. For any candidate future token $y_m$ with $n < m \leq N$, if $y_m$ appears in the top $K$ tokens when projecting $W_l[f]$ through the unembedding matrix, then $f$ is said to be a future-token encoding for $y_m$.

**Definition 2** (Precursor Influence)**.** For some $\alpha > 0$ and token position $t$, if subtracting a scaled decoding direction $\alpha W_l[f]$ from the residual stream at $(l, t)$ during the forward pass and regenerating the sequence from $t + 1$ causes

(i) a change in the next token $y_n$

(ii) a change in at least one intermediate token $y_{n+1}, \ldots, y_{m-1}$

(iii) removal of $y_m$ from the generated output

then the latent $f$ has a precursor influence on the future token $y_m$.

**Definition 3** (Planning)**.** A model is **planning** at position $(l, t)$, during the prediction of $y_n$, for a future token $y_m$ $(m > n > t)$ if there exists a latent $f$ at $(l, t)$ that is a future-token encoding (**FTE**) and has a precursor influence (**PI**) on the future token $y_m$.

We are not proposing that these definitions are complete and exhaustive. They are built considering the following working assumptions, and knowing these help us describe the boundaries of where these definitions will work.

1. If a model is planning for $y_m$during the prediction of $y_n$, then the circuit for predicting $y_n$will have latents related to $y_m$.
2. If $y_m$is in the top $K$ of logit lens [12] for a latent, then the latent is increasing the logit probability for $y_m$or “thinking” about $y_m$and it is “related to $y_m$”.
3. If planning for a future token occurs, then intermediate tokens are affected.
4. Negative steering a latent suppresses the token/concept it is related to from the activation space.

### 2.3 Feature Roles Induced by the Criteria

The *Future-Token Encoding* (**FTE**) and *Precursor Influence* (**PI**) criteria can be used to partition the set of $(l, f, t)$ triples into the following behaviorally distinct classes:

**Planning:** A planning feature satisfies *both* **FTE** and **PI** for some future token $y_m$ that is absent from the prompt. In other words, it stores a representation of $y_m$ and exerts an early causal influence that shapes the intermediate trajectory towards $y_m$. Removing the feature at the point where it first activated will prevent the token $y_m$ from being generated, and usually steers the generation down a semantically unrelated path.

**Improvisation:** A feature satisfies **FTE** for some $y_m$ but not **PI**; it only exerts a causal influence at the position just before $y_m$ is generated. In other words, steering or ablating that feature right before $y_m$ can change the next token, but doing the same at an earlier position will not change $y_m$ or any of the intermediate tokens leading up to it.

**Neither:** A feature fails **FTE** for every future token. This does not mean that the feature is not important; it could be encoding computations that keep the language model "on track" without explicitly referencing a future goal. These can include local syntax, formatting, short-range semantics, discourse markers, duplicate-token detectors, or many others.

**Can't Say:** This category represents scenarios that could be ambiguous, meaning the existence of a causal effect need not be interpreted as planning behavior.

- *Overlap with Prompt:* The goal token $y_m$ already appears earlier or ties another future token in the Logit Lens ranking. Even if **FTE** is satisfied, it is unclear whether this is just attending to a token in the prompt or planning for a future token.
- *Out-of-Distribution Steering:* When steering at an earlier position results in degenerate or nonsensical outputs, it is unclear whether to consider this to be the same as suppressing the planning mechanism. Hence even though the feature technically satisfies *PI*, we do not label it as a planning feature.

Appendix C includes examples and potential strategies for identifying planning and improvisation in these scenarios. For this work, we exclude these cases from all quantitative metrics.

### 2.4 Identifying Planning at Scale

For any model and prompt, we could apply Definition 3 to every $(l, f, t)$ triple and label them as PLAN or IMPROV. However, this is usually infeasible in practice. For example, Gemma-2-2B would require 26 layers $\times$ $16K$ latents $\times$ 100 tokens $\approx 4.2 \times 10^6$ tests per prompt. Indeed, many prior works on planning in LLMs focus on a single prompt or a handful of prompts.

We provide a pipeline to trim the search space by a factor of $\sim 10^4$ while preserving almost all genuine planning positions. An overview of our detection pipeline (circuit discovery to FTE to PI) is shown in Fig. 1.

**Step 0 Circuit Discovery:** Because **PI** already requires a causal effect on the next token $y_n$, we first isolate the sparse feature circuit that *explains* the prediction of $y_n$. Starting with the latents that have the highest indirect causal effect, we build the smallest set of $(l, f, t)$ triples $\mathcal{C}$ that can recover the original logit distribution $P_{\text{model}}(y_n)$ by at least $60\%$ when all other $(l, f, t)$ triples are zero-ablated. Empirically we find that $|\mathcal{C}| \in [2 \times 10^4, 3 \times 10^4]$, which is represents a decrease by a factor of $150\times$.

**Step 1 Future-Token Encoding Filter:** Apply **FTE** to every triple $(l, f, t) \in \mathcal{C}$ with $t < n$. Keep a triple only if its Logit-Lens top-$K$ contains some future token $y_m$, otherwise label it as NEITHER. Triples that share the same $(l, t)$ and point to the same $y_m$ are merged into a *cluster* $S$. In our experiments, each cluster contained on average $\sim$50 $(l, f, t)$ triples.

**Step 2 Cluster-Level Precursor Influence Check:** Steering the whole cluster at once is $\sim 50\times$ cheaper than steering its individual members. We subtract $\alpha \sum_{(l,f,t)\in S} W_l[f]$ when predicting $y_n$ for a range of $\alpha$ values. If **PI** is satisfied for the target $y_m$ without a degenerate output, the cluster is kept as a PLAN candidate; otherwise the whole cluster is considered a candidate for IMPROV or CAN'T SAY for nonsensical generations. Note that all $(l, f, t)$ triples inside $S$ satisfy **FTE** by construction, so NEITHER cannot occur here.

**Step 3 Earliest-Moment Search:** For within surviving clusters, greedily walk backwards through the positions where $S$ is active, ablating one triple at a time until PI fails. The last triple whose removal still deletes $y_m$ is recorded as the first backward-planning moment. Other $(l, f, t)$ triples in $S$ that satisfy both **FTE** and **PI** are also labeled PLAN.

**Step 4 Improvisation Check:** We rerun Step 2 for all $(l, f, t)$ that are not already labeled as PLAN but with $y_m$ as the next token. Any $(l, f, t)$ triple that has a causal effect on $y_m$ without satisfying **PI** for any of the previous tokens is labeled as IMPROV. For all $y_m$ already present in the input prompt, we also assign CAN'T SAY. The remaining are labeled as NEITHER.

Step 0 focuses solely on $y_n$ because any feature that fails to influence the next token being predicted *cannot* satisfy **PI**. Following from Lindsey et al. [9], we use cluster-level steering in Step 2 to amortize compute since discarding even two clusters early saves $\sim 100$ individual **PI** checks later. In the next section, we will empirically evaluate the above pipeline on real-world data.

# 3 Empirical Evaluation

We empirically evaluate our detection framework on the BASE and INSTRUCT versions of Gemma-2-2B [13]. We used TopK SAEs trained on `MLP_out` from the GemmaScope suite [14]. These SAEs are trained on the outputs of each MLP block before RMSNorm is applied.

Our analysis consists of three main components. We first provide a motivating example of our criteria/pipeline on several rhyming-couplet tasks (§3.1) to give a direct comparison to prior work Lindsey et al. [9]. We then demonstrate our detection framework on several programming tasks (§3.2). Finally, we provide a comparative analysis of planning in BASE vs. INSTRUCT models (§3.3).

## 3.1 Planning in Poems

We now revisit the example in §2.1 to evaluate our criteria and pipeline. Lindsey et al. [9] showed that Claude 3.5 Haiku activates latent features for candidate rhyme words such as *habit* and *rabbit* at the end of the first line (`'\n'`), six tokens before the model predicts the second rhyme - *"rabbit"*. When the same prompt, *"A rhyming couplet:*`\n` He saw a carrot and had to grab it, `\n''`, is given to Gemma-2-2B INSTRUCT, it completes it with *"A tasty treat, a crunchy habit."*

Running our FTE + PI pipeline (§2.4) over every intermediate prediction (from $y_1$ = "A" through $y_6$ = "tasty") reveals that *no circuit satisfies both criteria*. In other words, Gemma shows *no evidence of planning* during this poem generation. A binary verdict alone does not illuminate the model's internal strategy, so we manually inspected the full circuit for predicting "habit"($\sim$ 26k latents, $\geq 60\%$ logit recovery), similar to the setup of Lindsey et al. [9]. The circuit contains two distinct groups of latents: one that activates on phoneme-level tokens (e.g. "/t/", "/et/"), and another that activates on compulsion tokens(e.g. "had to grab", "must"). Latents writing to "habit" only become causally relevant at the *final* token. Lindsey et al. [9] also demonstrated that negative steering on the "habit" latents caused a change in the intermediate tokens, which didn't happen for this circuit in Gemma. Thus, our working hypothesis is one of *improvisation*: local phonetic and thematic cues combine late to select the rhyme, rather than a plan propagated forward from line one.

Differences in planning are expected given variations in architecture, scale, and training data. Our semi-automated pipeline surfaces those discrepancies, providing a systematic lens for future work on how modeling choices shape emergent planning behavior. We now move to evaluating the pipeline on code generations tasks, as coding tasks are well represented in the training data for Gemma 2 [15].

## 3.2 Planning in Code

We next execute and analyze the detection framework on several programming tasks. For this we consider the Mostly Basic Programming Problems (MBPP) dataset [16] . We filtered the tasks to include only those that the INSTRUCT model solves correctly, picking the first 60 for analysis. We then run the pipeline on this set of tasks. In the following, we provide a selection of case studies where the model exhibits planning by our criteria.

### 3.2.1 Sorting a list of tuples.

This task involves sorting a list of tuples `subjectmarks` *in-place* by the second element of each tuple. The INSTRUCT model correctly solves the task by using a lambda function to key into the index 1 of the tuple for sorting: `subjectmarks.sort(key=lambda x: x[1])`.

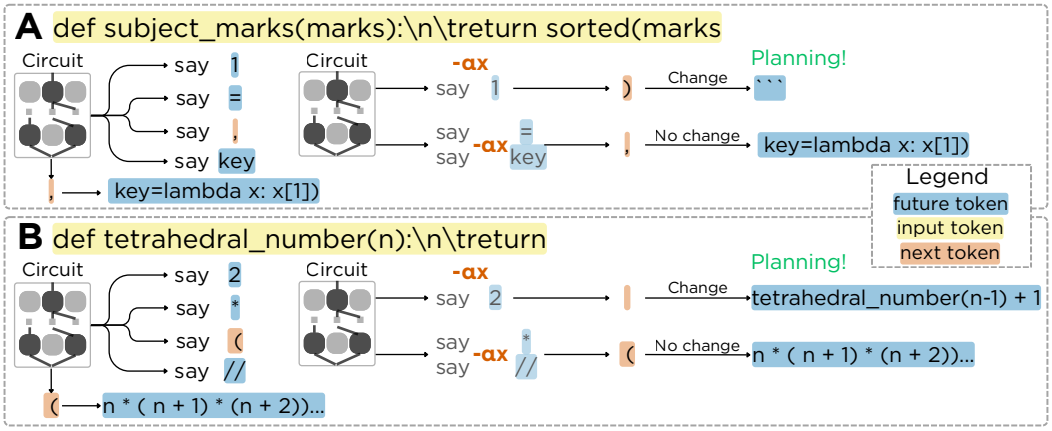


Figure 2: Planning in MBPP tasks. **(A) Sort tuples by the 2nd element:** While predicting the comma after "`sorted(marks`", a feature already promotes the future token `"1"` present in `key=lambda x: x[1]` (FTE). Suppressing it changes the generation to close the bracket instead (PI). **(B) $n$-th tetrahedral number:** While predicting the first parenthesis of the closed form, a feature encodes the later `"2"` in $(n{+}2)$ (FTE). Suppressing removes the plan: the model drifts to a recursive sketch (PI).

*Planning evidence.* During prediction of the comma ($y_n = 297$), the model is already planning for the future token `"1"` ($y_m = 305$) as early as ($\ell{=}0,\ t{=}294$) `"sorted"`. The earliest feature responsible writes in the direction of `"1"`, ranking it first among top-10 logits (FTE). Suppressing (negative steering) this feature flips the next token from the comma to a closing parenthesis (*i*). The generation just adds a newline and ends the function (*ii*), and `"1"` never appears in the continuation (*iii*). More intuitively, the model outputs the comma because it is planning to emit `"1"` later (it must sort by the second element of each tuple). Suppressing `"1"` features at this position causes the model to just close the bracket. The steered generation fails the unit test. See Fig. 2A for a schematic of this example (Appendix §B.1).

### 3.2.2 Computing the $n$-th tetrahedral number.

This task involves computing the $n$-th tetrahedral number $T(n)$. The INSTRUCT model correctly solves the task by using the closed form `(n*(n+1)*(n+2))//6` after handling small $n$, satisfying the tests.

*Planning evidence.* During prediction of the first opening bracket of the tetrahedral number formula ($y_n{=}180$), the model is already planning for the future token `"2"` as early as ($\ell{=}0,\ t{=}18$) (`"find"`). The earliest feature responsible writes in the direction of `"2"`, ranking it among the top-10 logits (FTE). Suppressing (negative steering) this feature switches the model from the closed form to the recursive update `tetrahedral_number(n-1)+1`: (*i*) `"tet"` is predicted as the next token instead of the opening parenthesis, (*ii*) the predictions after `"tet"` complete the recursive call, and (*iii*) `"2"` never appears (PI). This generation fails the unit tests. Intuitively, the model places the parenthesis because it is planning to emit the `"2"` needed for the $\times(n+2)$ factor; removing that plan pushes it back to a simpler recursive sketch that does not pass the tests. See Fig. 2**B** for the corresponding schematic (Appendix §B.2).

### 3.2.3 Forming the maximum number from digits.

Given a list of digits, the task is to return the largest possible integer formed by concatenating all elements from the list. The INSTRUCT model correctly solves the task by sorting the list in descending order and traverses it with a for-loop whose index variable is `i`.

*Planning evidence.* During prediction of `"digits"` (the first non-docstring token, $y_n{=}191$), the model is already planning for the future token `"sort"` as early as ($\ell{=}17,\ t{=}190$), which is the first tab (`"\t"`) after the docstring. The earliest feature responsible writes toward `"sort"`, placing it in the top-10 logits (FTE). Suppressing (or negatively steering) the `"sort"` feature at the `"digits"` token position (*i*) flips the next token to `max`, (*ii*) yields a program that instead begins `max_num =` and

fails all hidden tests, and (*iii*) removes `"sort"` entirely from the continuation (PI). Intuitively, the model commits to `"digits"` because it is planning to immediately call `sort`; without that plan, it never orders the digits and thus cannot construct the maximum number. See Fig. 4 for the schematic (Appendix §B.3).

#### 3.2.4 Examples of "can't say" cases.

**Divisible tuples.** The task is to return only those tuples whose elements are all divisible by a given number $k$. The baseline generation correctly completes the comprehension `"== 0 for element in tup)"` and appends matching tuples before returning, satisfying the tests. For the prediction of the next token `"=="`, we find features writing to the direction of `"for"`, which is a future token. With the steering token `"for"` (coeff $-80$), the output collapses around the divisor check into something like `", k"):` and loses the generator expression, yielding a syntactically invalid snippet. This satisfies both FTE and PI, but because `for` is also in the input we label it as "can't say" (Appendix §C.1).

**Largest number from digits.** The task is to rearrange a list of digits to form the maximum possible integer. The baseline sorts the list and builds the number by concatenating digits in reverse order (e.g., `max_num += str(digits[n-i-1])`) and returns `int(max_num)`, which passes the tests. For the prediction of the next token `"num"`, we find features writing to `"-"`, which is a future token and is not in the input. Under steering with the token `"-"` (coeff $-80$), the model veers into nonsense (e.g., `digits = len(digits)` followed by stray triple-quoted lines), so the steered output is degenerate (Appendix §C.2).

**Tetrahedral number.** This is the same task as §3.2.2, but for a different forward pass. With `else` being the next token.

With the steering token `"("` (coeff $-100$), the generation devolves into a stream of "the/The" without code or logic, so the steered output is degenerate. Note that the steered token `"("` is in the original prompt already, in the function signature and assertions. (Appendix §C.3).

Across all three case studies the same pattern emerges: an SAE-cluster that (i) linearly encodes a distant goal token and (ii) causally steers multiple intermediate tokens is necessary for the model's success, thereby validating our planning labels. Overall, our pipeline identifies the INSTRUCT model as either planning or improvising on 24 out of 60 tasks.

### 3.3 Comparing Base and Instruction-Tuned Models

Although base models are trained to predict the next token, post-training methods such as instruction tuning and RL introduce multi-step or goal-oriented objectives, and we hypothesize that these post-training methods result in stronger planning behavior. In this section, we explore this hypothesis and compare the planning behaviors between the INSTRUCT and BASE models.

Table 1: Pass rates by task subset for the INSTRUCT and BASE Gemma-2-2B models. Values are percentages; $n$ denotes the number of tasks evaluated in each subset.

| Task subset | INSTRUCT model | BASE model |
|---|---|---|
| Planning tasks ($n = 13$) | 100% | 54% |
| Improvisation tasks ($n = 11$) | 100% | 100% |

For this comparison, we focus on the 24 MBPP tasks identified in Section 3.2 and described in Table 1, where our detection pipeline classified whether the INSTRUCT model was planning or improvising. We evaluate the BASE model on our chosen subset of MBPP that the INSTRUCT model solves, and compare the performance on the IMPROV cases vs. the PLAN cases.

We find that BASE solves all tasks where INSTRUCT was improvising, but only 54% (7 out of 13) of tasks where INSTRUCT was planning. The BASE model's ability to solve these planning tasks suggests two possibilities: either BASE already possesses some planning capabilities without instruction tuning, or it can solve these tasks without planning.

To examine these hypotheses, we apply our planning detection pipeline to the BASE model on the same planning tasks. We find that in many cases BASE is still capable of planning, but there are

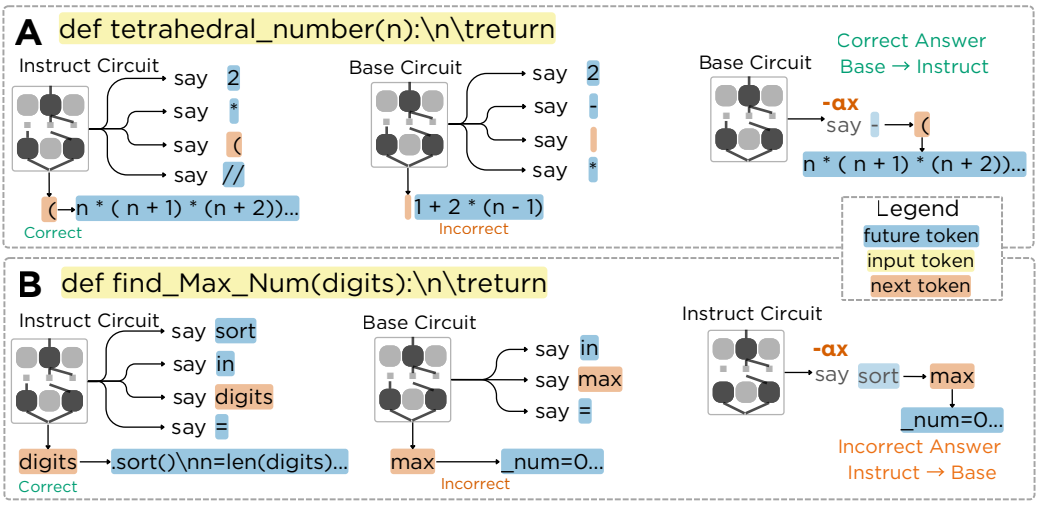


Figure 3: Instruction tuning refines plan selection. **(A) Competing plans (Tetrahedral number)**: Both models plan toward `"2"`, but BASE also plans toward an alternative `"-"` path that yields an incorrect closed form. Suppressing `"-"` for BASE removes the competing plan and recovers the correct solution, matching INSTRUCT. **(B) Incorrect Target (Largest number)**: INSTRUCT plans to `sort` the `digits`, but BASE plans toward `max` and never sorts, leading to failure. Suppressing INSTRUCT's `sort` plan reproduces the failed BASE trajectory. Diagrams highlight planning features; $-\alpha x$ indicates steering.

distinct failure modes that result in incorrect answers. We identify two primary failure patterns in the following subsections.

### 3.3.1 Competing Plans

In the Nth Tetrahedral Number task covered in §3.2.2 (Fig. 3A, Appendix §B.2), BASE exhibits planning by targeting the correct token `"2"`, but also plans for an incorrect alternative `"-"`. When the model follows the incorrect plan, it generates the wrong formula (`1+2(n-1)`). In contrast, INSTRUCT focuses solely on the correct plan and produces the right solution (`n*(n+1)*(n+2)/6`).

Given these results, one potential explanation is that *the* BASE *model is still planning, but the plan is not yet specific enough*. If this were true, suppressing the wrong token that the model is also planning for should bring the behavior of BASE closer to that of INSTRUCT. Indeed we find that suppressing the `"-"` token features causes BASE to return the correct formula.

### 3.3.2 Incorrect Target

In the task for forming largest number from list covered in §3.2.3 (Fig. 3B, Appendix §B.3), BASE is planning for the `"max"` token unlike INSTRUCT which is planning for the `"digits"` token. However, unlike in the previous example, BASE does not have `"digits"` as a potential plan, and therefore fails return a correct answer. Suppressing the `"digits"` feature in INSTRUCT leads it to return the same incorrect answer that BASE does.

Overall, these results suggest that base models still exhibit planning behavior, and instruction tuning is likely not the source of planning per se. However, instruction tuning can improve performance on planning tasks by helping the model select the right tokens to target.

## 4 Conclusion

In this work, we introduced a general, falsifiable definition of planning in language models that generalizes and extends insights from prior case studies. We operationalized this definition through two criteria, Future-Token Encoding (**FTE**) and Precursor Influence (**PI**), and implemented a semi-

automated pipeline to detect them. Applying this framework to the base and instruction-tuned versions of Gemma-2-2B on various MBPP code generation tasks, we demonstrated that:

- **Planning is not universal.** The model solves some tasks by improvising and others by planning, and we found no clear rule governing which strategy is used. Planning does not appear to be task-specific either; Gemma-2-2B improvises on a poem generation task where Claude 3.5 Haiku was shown to plan [9], though both models still generated valid poems.
- **Planning can be done poorly.** We found cases where the model deliberately planned toward incorrect answers or selected incorrectly among multiple competing plans.
- **Instruction tuning refines planning behavior but does not create it.** Both base and instruction tuned models are capable of planning, but it is possible that instruction tuning helps with choosing between competing plans or filtering out incorrect plans.

### 4.1 Limitations

**SAEs** Our analysis was conducted using SAEs trained on the Gemma-2-2B base model but applied to the instruction-tuned version of Gemma-2-2B. While this approach is supported by prior work [17], it may introduce some mismatch in representation. Furthermore, we focused only on MLP-attached SAEs, inspired by their interpretability in prior work [9]. That said, our detection pipeline and criteria are general and can be extended to other types of SAEs and representation spaces.

**Polysemantic latents** We found cases where some latents satisfy both FTE and PI, but upon manual investigation, we see that only one of the top 10 tokens is a future token and the others seem unrelated to the task. This is probably because the latent is polysemantic. We could potentially mitigate this by requiring stricter criteria such as a minimum threshold on autointerp scores.

**Scaling to larger models and broader datasets.** Our study focuses on the base and instruct versions of Gemma-2-2B, mainly due to the availability of SAEs for all layers [14]. We plan to apply our detection framework to larger models in the Gemma family to understand how planning capabilities emerge as a function of scale. Additionally, our experiments have thus far focused on the MBPP dataset and the poem generation prompt from Lindsey et al. [9]. Extending this analysis to more challenging and diverse benchmarks could reveal deeper insights into planning behavior.

### 4.2 Future Work

**Understanding edge and "can't say" cases.** A significant portion of our effort was spent on ambiguous or edge cases, where labels could not be clearly assigned. More details on such cases can be found in Appendix C; investigating these further could refine our definition and improve detection.

**Automating offline detection** We found cases where steered generations satisfied our criteria for Precursor Influence, but the generated text itself was degenerate and nonsensical. It is not clear to us if this is an instance of planning behavior switching off or if the intervention pushed the model out of distribution. Setting thresholds for repeating tokens and perplexity could potentially help resolve this.

**Online detection** This paper focuses on offline detection, i.e., detecting planning after the sequence is generated. with the knowledge of future tokens. However, we believe it is possible to extend our approach to detect planning at test time, where we have no knowledge of the future tokens. For example, at each token prediction we can find latents that write to tokens that are not present in the input; these become candidates for planning.

## References

[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners.

In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[2] Augustus Odena, Charles Sutton, David Martin Dohan, Ellen Jiang, Henryk Michalewski, Jacob Austin, Maarten Paul Bosma, Maxwell Nye, Michael Terry, and Quoc V. Le. Program synthesis with large language models. In *n/a*, page n/a, n/a, 2021. n/a.

[3] Koyena Pal, Jiuding Sun, Andrew Yuan, Byron Wallace, and David Bau. Future lens: Anticipating subsequent tokens from a single hidden state. In *Proceedings of the 27th Conference on Computational Natural Language Learning (CoNLL)*, page 548–560. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.conll-1.37. URL http://dx.doi.org/10.18653/v1/2023.conll-1.37.

[4] Nicholas Pochinkov, Angelo Benoit, Lovkush Agarwal, Zainab Ali Majid, and Lucile Ter-Minassian. Extracting paragraphs from llm token activations, 2024. URL https://arxiv.org/abs/2409.06328.

[5] Wilson Wu, John Xavier Morris, and Lionel Levine. Do language models plan ahead for future tokens? In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BaOAvPUyBO.

[6] Gian-Carlo Pascutto and Gary Linscott. Leela chess zero. URL http://lczero.org/.

[7] Erik Jenner, Shreyas Kapur, Vasil Georgiev, Cameron Allen, Scott Emmons, and Stuart Russell. Evidence of learned look-ahead in a chess-playing neural network, 2024. URL https://arxiv.org/abs/2406.00877.

[8] Thomas Bush, Stephen Chung, Usman Anwar, Adrià Garriga-Alonso, and David Krueger. Interpreting emergent planning in model-free reinforcement learning, 2025. URL https://arxiv.org/abs/2504.01871.

[9] Jack Lindsey, Wes Gurnee, Emmanuel Ameisen, Brian Chen, Adam Pearce, Nicholas L. Turner, Craig Citro, David Abrahams, Shan Carter, Basil Hosmer, Jonathan Marcus, Michael Sklar, Adly Templeton, Trenton Bricken, Callum McDougall, Hoagy Cunningham, Thomas Henighan, Adam Jermyn, Andy Jones, Andrew Persic, Zhenyi Qi, T. Ben Thompson, Sam Zimmerman, Kelley Rivoire, Thomas Conerly, Chris Olah, and Joshua Batson. On the biology of a large language model. *Transformer Circuits Thread*, 2025. URL https://transformer-circuits.pub/2025/attribution-graphs/biology.html.

[10] Anthropic. Claude haiku. https://www.anthropic.com/claude/haiku, 2024. Large language model.

[11] Robert Huben, Hoagy Cunningham, Logan Riggs Smith, Aidan Ewart, and Lee Sharkey. Sparse autoencoders find highly interpretable features in language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=F76bwRSLeK.

[12] Nostalgebraist. Interpreting gpt: the logit lens. https://www.lesswrong.com/posts/logit-lens, 2020.

[13] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, Johan Ferret, Peter Liu, Pouya Tafti, Abe Friesen, Michelle Casbon, Sabela Ramos, Ravin Kumar, Charline Le Lan, Sammy Jerome, Anton Tsitsulin, Nino Vieillard, Piotr Stanczyk, Sertan Girgin, Nikola Momchev, Matt Hoffman, Shantanu Thakoor, Jean-Bastien Grill, Behnam Neyshabur, Olivier Bachem, Alanna Walton, Aliaksei Severyn, Alicia Parrish, Aliya Ahmad, Allen Hutchison, Alvin Abdagic, Amanda Carl, Amy Shen, Andy Brock, Andy Coenen, Anthony Laforge, Antonia Paterson, Ben Bastian, Bilal Piot, Bo Wu, Brandon Royal, Charlie Chen, Chintu Kumar, Chris Perry, Chris Welty, Christopher A. Choquette-Choo, Danila Sinopalnikov, David Weinberger, Dimple Vijaykumar, Dominika Rogozińska, Dustin Herbison, Elisa Bandy, Emma Wang, Eric Noland, Erica Moreira, Evan Senter, Evgenii Eltyshev, Francesco Visin, Gabriel Rasskin, Gary Wei, Glenn Cameron, Gus Martins, Hadi Hashemi, Hanna Klimczak-Plucińska, Harleen Batra, Harsh Dhand, Ivan Nardini, Jacinda Mein, Jack Zhou, James Svensson, Jeff Stanway, Jetha Chan, Jin Peng Zhou, Joana Carrasqueira, Joana Iljazi, Jocelyn Becker, Joe Fernandez, Joost van Amersfoort, Josh Gordon, Josh Lipschultz, Josh Newlan, Ju yeong Ji,

Kareem Mohamed, Kartikeya Badola, Kat Black, Katie Millican, Keelin McDonell, Kelvin Nguyen, Kiranbir Sodhia, Kish Greene, Lars Lowe Sjoesund, Lauren Usui, Laurent Sifre, Lena Heuermann, Leticia Lago, Lilly McNealus, Livio Baldini Soares, Logan Kilpatrick, Lucas Dixon, Luciano Martins, Machel Reid, Manvinder Singh, Mark Iverson, Martin Görner, Mat Velloso, Mateo Wirth, Matt Davidow, Matt Miller, Matthew Rahtz, Matthew Watson, Meg Risdal, Mehran Kazemi, Michael Moynihan, Ming Zhang, Minsuk Kahng, Minwoo Park, Mofi Rahman, Mohit Khatwani, Natalie Dao, Nenshad Bardoliwalla, Nesh Devanathan, Neta Dumai, Nilay Chauhan, Oscar Wahltinez, Pankil Botarda, Parker Barnes, Paul Barham, Paul Michel, Pengchong Jin, Petko Georgiev, Phil Culliton, Pradeep Kuppala, Ramona Comanescu, Ramona Merhej, Reena Jana, Reza Ardeshir Rokni, Rishabh Agarwal, Ryan Mullins, Samaneh Saadat, Sara Mc Carthy, Sarah Cogan, Sarah Perrin, Sébastien M. R. Arnold, Sebastian Krause, Shengyang Dai, Shruti Garg, Shruti Sheth, Sue Ronstrom, Susan Chan, Timothy Jordan, Ting Yu, Tom Eccles, Tom Hennigan, Tomas Kocisky, Tulsee Doshi, Vihan Jain, Vikas Yadav, Vilobh Meshram, Vishal Dharmadhikari, Warren Barkley, Wei Wei, Wenming Ye, Woohyun Han, Woosuk Kwon, Xiang Xu, Zhe Shen, Zhitao Gong, Zichuan Wei, Victor Cotruta, Phoebe Kirk, Anand Rao, Minh Giang, Ludovic Peran, Tris Warkentin, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, D. Sculley, Jeanine Banks, Anca Dragan, Slav Petrov, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Sebastian Borgeaud, Noah Fiedel, Armand Joulin, Kathleen Kenealy, Robert Dadashi, and Alek Andreev. Gemma 2: Improving open language models at a practical size, 2024. URL https://arxiv.org/abs/2408.00118.

[14] Tom Lieberum, Senthooran Rajamanoharan, Arthur Conmy, Lewis Smith, Nicolas Sonnerat, Vikrant Varma, János Kramár, Anca Dragan, Rohin Shah, and Neel Nanda. Gemma scope: Open sparse autoencoders everywhere all at once on gemma 2, 2024. URL https://arxiv.org/abs/2408.05147.

[15] Google. Gemma 2 Model Card. https://ai.google.dev/gemma/docs/core/model_card_2, 2025. Training-dataset section, accessed 2 Aug 2025.

[16] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

[17] Connor Kissane, Robert Krzyzanowski, Arthur Conmy, and Neel Nanda. Saes (usually) transfer between base and chat models, 2024. URL https://www.lesswrong.com/posts/fmwk6qxrpW8d4jvbd/saes-usually-transfer-between-base-and-chat-models. LessWrong, published July 18, 2024.

[18] Samuel Marks, Can Rager, Eric J Michaud, Yonatan Belinkov, David Bau, and Aaron Mueller. Sparse feature circuits: Discovering and editing interpretable causal graphs in language models. *arXiv preprint arXiv:2403.19647*, 2024.

[19] Jesse Vig, Sebastian Gehrmann, Yonatan Belinkov, Sharon Qian, Daniel Nevo, Yaron Singer, and Stuart Shieber. Investigating gender bias in language models using causal mediation analysis. *Advances in neural information processing systems*, 33:12388–12401, 2020.

[20] Matthew Finlayson, Aaron Mueller, Sebastian Gehrmann, Stuart Shieber, Tal Linzen, and Yonatan Belinkov. Causal analysis of syntactic agreement mechanisms in neural language models. *arXiv preprint arXiv:2106.06087*, 2021.

[21] Judea Pearl. Direct and indirect effects. In *Probabilistic and causal inference: the works of Judea Pearl*, pages 373–392. 2022.

[22] Neel Nanda. Attribution patching: Activation patching at industrial scale. *URL: https://www.neelnanda. io/mechanistic-interpretability/attribution-patching*, 2023.

[23] Aaquib Syed, Can Rager, and Arthur Conmy. Attribution patching outperforms automated circuit discovery, 2023. URL https://arxiv.org/abs/2310.10348.

[24] János Kramár, Tom Lieberum, Rohin Shah, and Neel Nanda. Atp*: An efficient and scalable method for localizing llm behaviour to components. *arXiv preprint arXiv:2403.00745*, 2024.

[25] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *International conference on machine learning*, pages 3319–3328. PMLR, 2017.

[26] Samuel Marks, Can Rager, Eric J. Michaud, Yonatan Belinkov, David Bau, and Aaron Mueller. Sparse feature circuits: Discovering and editing interpretable causal graphs in language models, 2025. URL https://arxiv.org/abs/2403.19647.

# A Background

We review the three ingredients our method builds on: sparse autoencoders, causal-influence localization, and prior work on planning in neural networks.

## A.1 Sparse Autoencoders

Sparse autoencoders (SAEs) have gained popularity as an unsupervised interpretability method for analyzing activations of large language models. An SAE generally consists of an encoder-decoder structure: the encoder transforms the original activations into a higher-dimensional but sparse latent representation, while the decoder reconstructs the original activations from this sparse representation.

We use SAEs from the GemmaScope suite (Lieberum et al. [14]). From this suite, we used TopK SAEs trained on MLP_out. These SAEs are trained on the outputs of each MLP block, before RMSNorm is applied.

## A.2 Causal influence: activation & attribution patching

We follow Marks et al. [18] for notations and approximations for circuit discovery with SAEs.

**Indirect effects** Following Vig et al. [19], Finlayson et al. [20], let $m$ be any scalar metric of the forward pass (e.g. $-\log P_\theta(y_n)$) and let $a$ be an internal activation. For a *clean / patch* input pair $(x_{\text{clean}}, x_{\text{patch}})$ we measure the *indirect effect* (IE) [21] of $a$ on $m$ as

$$\text{IE}\big(m; a; x_{\text{clean}}, x_{\text{patch}}\big) \;=\; m\big(x_{\text{clean}} \,\big|\, \text{do}(a{=}a_{\text{patch}})\big) - m\big(x_{\text{clean}}\big), \tag{1}$$

where the do-operator fixes $a$ to its value $a_{\text{patch}}$ taken from the patched run.

However, computing (1) for every candidate $a$ is expensive, so we adopt two gradient-based approximations. *Attribution patching* [22, 23, 24] linearizes IE with a first-order Taylor expansion, needing only *two* forward passes and *one* back-propagation:

$$\widehat{\text{IE}}_{\text{AP}} = \nabla_a m\big|_{a=a_{\text{clean}}} \big(a_{\text{patch}} - a_{\text{clean}}\big). \tag{2}$$

**Integrated gradients (IG)** [25] trades extra compute for a tighter fit. Using $N{=}10$ evenly spaced interpolation points $\alpha \in [0, 1]$ we form

$$\widehat{\text{IE}}_{\text{IG}} = \frac{1}{N}\sum_{k=1}^{N} \nabla_a m\Big|_{a=a_{\text{clean}}+\frac{k}{N}\left(a_{\text{patch}}-a_{\text{clean}}\right)} \big(a_{\text{patch}} - a_{\text{clean}}\big), \tag{3}$$

which markedly improves accuracy.

**Single-prompt variant.** When only one prompt is available we replace $(x_{\text{clean}}, x_{\text{patch}})$ with $(x, x)$ and set $a_{\text{patch}} = 0$, i.e. we measure the drop in $m$ under *zero-ablation* of $a$; the same formulas (2)–(3) apply after substituting $a_{\text{patch}} \leftarrow 0$.

## A.3 Planning in neural networks

**Predicting future tokens (fixed $k$).** Early work asked whether a single intermediate representation linearly encodes the *final* logits $k$ steps ahead. Pal et al. [3] trained an affine probe that can predict the top-$k$ logits four tokens in the future in GPT-2, but only for some layers. Pochinkov et al. [4] extend this to paragraph-level topics, showing that the newline token between paragraphs already carries topical information. Wu et al. [5] repeat the experiment across model scales and find that small models exhibit little signal, whereas larger models show modest top-token predictability. In all cases the horizon $k$ is fixed by the probe designer.

**Learned look-ahead in games and RL.** Outside language, neural agents sometimes plan several moves ahead. Jenner et al. [7] detect representations of optimal next moves up to three ply ahead in Leela ChessZero [6] by training chess-specific linear heads. Bush et al. [8] identify state vectors in a Sokoban-playing agent that encode the sequence of box moves needed to solve the puzzle, again with a fixed look-ahead. These studies reinforce the possibility of learned planning but remain task-specific and horizon-bound.

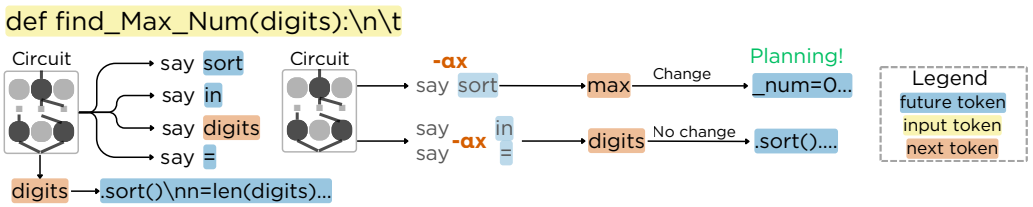


Figure 4: Forming the max number from digits: While predicting the the first non-docstring "`digits`", a feature already promotes the future token `"sort"` present in `digits.sort()\n` (FTE). Suppressing it changes the generation to not sort instead and start with `max_num = 0` (PI).

**Variable-horizon planning.** The poem-rhyme case study of Lindsey et al. [9] shows that a large language model (Claude 3.5 Haiku) stores candidate rhyme words an arbitrary number of tokens in advance and that ablating this latent collapses the rhyme. This example motivates the formal criteria we adopt in Section 2.

**Gap addressed by our work.** All prior studies either assume a fixed horizon or require task-specific probes. Our criteria work for any distance $m - n \geq 1$ and rely only on model-intrinsic SAEs plus causal steering.

# B Additional details for "planning" cases

This appendix presents three representative "planning" cases. For each case, we show:

- the **Prompt Prefix** (truncated to the noted token),
- the **Baseline Generation** continuation,
- the **Steering Token** and its **Coefficient**, and
- the resulting **Steered Continuation**.

All snippets below are exact text captures.

## B.1 Sorting list of tuples.

**Prompt Prefix (up to token 297):**

```
<bos>You are an expert Python programmer, and here is your task: Write a function
to sort a list of tuples using the second value of each tuple. Your code should
pass these tests:

assert subject_marks([('English', 88), ('Science', 90), ('Maths', 97), ('Social
sciences', 82)])==[('Social sciences', 82), ('English', 88), ('Science', 90),
('Maths', 97)]
assert subject_marks([('Telugu',49),('Hindhi',54),('Social',33)])==([('Social',33),
('Telugu',49),('Hindhi',54)])
assert
subject_marks([('Physics',96),('Chemistry',97),('Biology',45)])==([('Biology',45),
('Physics',96),('Chemistry',97)])
Write your code below starting with "‘‘‘python" and ending with "‘‘‘".
‘‘‘python
def subject_marks(marks):
    """
    Sorts a list of tuples by the second value of each tuple.

    Args:
        marks: A list of tuples, where each tuple represents a subject and its
corresponding mark.

    Returns:
```

```
        A new list of tuples, sorted by the second value of each tuple.
    """
    return sorted(marks
```

**Baseline Generation:**

```
, key=lambda x: x[1])
```

**Steering Token and Coefficient:** ‘1’, Coeff -80

**Steered Continuation:**

```
)
```

### B.2 Computing the $n$-th tetrahedral number.

**Prompt Prefix (up to token 180):**

```
<bos>You are an expert Python programmer, and here is your task: Write a function
to find the nth tetrahedral number. Your code should pass these tests:

assert tetrahedral_number(5) == 35
assert tetrahedral_number(6) == 56
assert tetrahedral_number(7) == 84
Write your code below starting with "‘‘‘python" and ending with "‘‘‘‘".
‘‘‘python
def tetrahedral_number(n):
    """
    Calculates the nth tetrahedral number.

    Args:
        n: The position of the tetrahedral number in the sequence.

    Returns:
        The nth tetrahedral number.
    """
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return
```

**Baseline Generation:**

```
 (n * (n + 1) * (n + 2)) // 6
```

**Steering Token and Coefficient:** ‘1’, Coeff -80

**Steered Continuation:**

```
 tetrahedral_number(n-1) + 1
```

### B.3 Forming the maximum number from digits.

**Prompt Prefix (up to token 311):**

```
<bos>You are an expert Python programmer, and here is your task: Write a python
function to find the largest number that can be formed with the given list of
digits. Your code should pass these tests:
```

```
assert find_Max_Num([1,2,3]) == 321
assert find_Max_Num([4,5,6,1]) == 6541
assert find_Max_Num([1,2,3,9]) == 9321
Write your code below starting with "‘‘‘python" and ending with "‘‘‘‘".
‘‘‘python
def find_Max_Num(digits):
    """
    Finds the largest number that can be formed with the given list of digits.

    Args:
        digits: A list of digits.

    Returns:
        The largest number that can be formed with the given digits.
    """
```

**Baseline Generation:**

```
digits.sort()
    n = len(digits)
    max_num = ''
    for i in range(n):
        max_num += str(digits[n-i-1])
    return int(max_num)
```

**Steering Token and Coefficient:** `'sort'`, `Coeff -60`

**Steered Continuation:**

```
max_num = 0
    for num in digits:
        if num > max_num:
            max_num = num
    return max_num
```

# C Details for "can't say" cases

This appendix presents three representative "can't say" cases. For each case, we show the same information as §B.

## C.1 Overlap with Prompt

**Prompt Prefix (up to token 311):**

```
<bos>You are an expert Python programmer, and here is your task: Write a function
to find tuples which have all elements divisible by k from the given list of
tuples. Your code should pass these tests:

assert find_tuples([(6, 24, 12), (7, 9, 6), (12, 18, 21)], 6) == [(6, 24, 12)]
assert find_tuples([(5, 25, 30), (4, 2, 3), (7, 8, 9)], 5) == [(5, 25, 30)]
assert find_tuples([(7, 9, 16), (8, 16, 4), (19, 17, 18)], 4) == [(8, 16, 4)]
Write your code below starting with "‘‘‘python" and ending with "‘‘‘‘".
‘‘‘python
def find_tuples(tuples_list, k):
    """
    Finds tuples in a list of tuples that have all elements divisible by k.

    Args:
        tuples_list: A list of tuples.
        k: The divisor.
```

```
    Returns:
        A list of tuples that have all elements divisible by k.
    """
    result = []
    for tup in tuples_list:
        if all(element % k
```

**Baseline Generation:**

```
 == 0 for element in tup):
            result.append(tup)
    return result
```

**Steering Token and Coefficient:** `'for'`, `Coeff -80`

**Steered Continuation:**

```
, k):
            result.append(tup)
    return result
```

### C.2 Out-of-Distribution Steering

**Prompt Prefix (up to token 223):**

```
<bos>You are an expert Python programmer, and here is your task: Write a python
function to find the largest number that can be formed with the given list of
digits. Your code should pass these tests:

assert find_Max_Num([1,2,3]) == 321
assert find_Max_Num([4,5,6,1]) == 6541
assert find_Max_Num([1,2,3,9]) == 9321
Write your code below starting with "‘‘‘python" and ending with "‘‘‘‘".
‘‘‘python
def find_Max_Num(digits):
    """
    Finds the largest number that can be formed with the given list of digits.

    Args:
        digits: A list of digits.

    Returns:
        The largest number that can be formed with the given digits.
    """
    digits.sort()
    n = len(digits)
    max_num = ''
    for i in range(n):
        max_
```

**Baseline Generation:**

```
num += str(digits[n-i-1])
    return int(max_num)
```

**Steering Token and Coefficient:** `'-'`, `Coeff -80`

**Steered Continuation:**

```
digits = len(digits)
        """
```

```
    """
    """
    """
    """
    """
    """
    """
    """
    """
    """
    """
    """
    """
    """
    """
    """
    """
    """
    """
    """
    """
    """
    """
    """
```

### C.3 Both Overlap and Out-of-Distribution

**Prompt Prefix (up to token 173):**

````
<bos>You are an expert Python programmer, and here is your task: Write a function
to find the nth tetrahedral number. Your code should pass these tests:

assert tetrahedral_number(5) == 35
assert tetrahedral_number(6) == 56
assert tetrahedral_number(7) == 84
Write your code below starting with "```python" and ending with "```".
```python
def tetrahedral_number(n):
    """
    Calculates the nth tetrahedral number.

    Args:
        n: The position of the tetrahedral number in the sequence.

    Returns:
        The nth tetrahedral number.
    """
    if n <= 0:
        return 0
    elif n == 1:
        return 1
````

**Baseline Generation:**

```

    else:
        return (n * (n + 1) * (n + 2)) // 6
```
```

**Steering Token and Coefficient:** `'(', Coeff -100`

**Steered Continuation:**

```
 the

        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
        The
```

# D Ethics Statement

While planning detection has potential beneficial applications for enhancing model controllability, it also raises ethical concerns. The ability to manipulate model outputs could be misused to bypass safety measures or to make models generate harmful content. We emphasize the importance of responsible use of these techniques and suggest the development of countermeasures to protect against potential misuse.

# E Artifacts

## E.1 MBPP Dataset

The original Mostly Basic Programming Problems (MBPP) dataset [16] features 974 python programming problems featuring a text description of a function, along with a set of unit tests that a model's generated code is supposed to pass. We filtered this dataset down to the 120 tasks that Gemma-2-2B solves correctly with deterministic sampling (temperature 0). The entire subset used for all analysis; no train/dev/test split required since we perform interpretability on fixed generations.

### E.1.1 Tasks

**Compliance** This dataset uses the Creative Commons Attribution 4.0 International (CC BY 4.0) license.

### E.2 Gemma-2-2B

We use Gemma-2-2B (2 billion parameters) – a decoder-only Transformer with 26 layers and RMSNorm pre- and post-normalization (see Team et al., 2024 for full architecture and training details).

**Compliance** Gemma-2 models are released under Google's commercially-friendly Gemma License, which permits model usage for research and evaluation purposes only.

### E.3 Sparse Autoencoders

We used code and sparse autoencoder weights (SAE) from the GemmaScope release, trained on the base Gemma-2-2B model. We 26 SAEs, one for each layer, trained to reconstruct the outputs of the MLP layers, before the post-RMSNorm is applied. These SAEs use the TopK activation function with $K = 32$, a latent dimension of $2048$ matching the MLP out dimension, and have an expansion factor of $8$ for a total of $16384$ features per layer.

**Compliance** The code and model checkpoints are distributed freely under the Apache 2.0 license.

## F Experimental Details

**Hardware and Compute** We used a single node of 4x NVIDIA A40 GPUs (48 GB VRAM). Total compute 250 GPU-hours across all experiments: Computing Sparse Feature Circuits using attribution patching took $10$ hours to run across all prompts and token generations. Computing precursor influence by steering clusters of features took the bulk of the compute with about 240 GPU-hours total.

**Algorithm Hyperparameters** We based our implementation for attribution patching from the source code from [26], code available here. The original algorithm uses clean and counterfactual pairs of prompts, whereas we use only clean prompts then perform zero ablations on intermediate representations.

The algorithm takes a metric $m$ to backpropagate through, a hyperparemeter $\tau$ for the metric, and number of integrated gradient steps $n$. We use the probability of the correct token $p(y_{\text{correct}})$ as the metric. We set $\tau = 0.60$, meaning we keep nodes that preserve the correct token's probability to be above 60%, and set $n = 10$.

## G Statement on the Usage of Generative AI

We used generative AI tools (e.g., GitHub Copilot and ChatGPT) to streamline routine coding tasks—such as writing data-loading scripts. In each case, all AI-suggested code was carefully reviewed, tested, and revised by the authors to ensure correctness and maintain consistent coding style. We used ChatGPT with search to generate high-level literature summaries that informed our reference list and contextual background, which were cross-checked against original papers before inclusion.