LM Assertions: Computational Constraints for Self-Refining Language Model Pipelines

Anonymous Author(s)
Affiliation
Address
email

Abstract

Chaining calls to language models (LMs) with composable modules is fueling a new way of programming, but ensuring LMs adhere to important constraints still requires heuristic prompt engineering. We introduce **LM Assertions**, a programming construct for expressing computational constraints that LMs should satisfy. LM assertions are an omni-construct that translates to multiple intriguing aspects of optimizing LM pipelines. For inference, they facilitate self-refinement by providing feedback on erroneous outputs. For in-context learning, LM assertions introduce demonstrations that adhere to arbitrary constraints and negative demonstrations that the LM must avoid. We report on four diverse case studies for text generation and find that LM Assertions improve compliance with imposed rules and downstream task performance, passing constraints from 0.0% to 98.0% and generating up to 166.6% more higher-quality responses.

1 Introduction

Language models (LMs) now power various applications, from conversational agents to writing assistants. However, the probabilistic nature of LMs often results in outputs that may not align with the domain's constraints or the larger pipeline in which the LM is used. To address this, researchers have explored various techniques, including applying constrained decoding [7, 8], exploring approaches for self-reflection and tree search [12, 18, 21], building domain-specific languages and like LMQL [2], or monitoring models with assertions and guardrails [9, 15] to steer LMs towards more controllable outputs.

Recently, several LM frameworks like LangChain [4] and DSPy [10, 11] provide developers with interfaces to build compound AI systems to encapsulate LM prompting pipelines [6, 17]. Some offer several features to control LM outputs, such as DSPy, which can optimize multi-stage prompts to maximize a target metric. However, such pipelines currently do not consider *arbitrary computational constraints* when instructing the LM to follow such constraints and to introspectively *self-refine outputs*. While some of this may be achieved via painstaking "prompt engineering" or other ad hoc guidance strategies, such efforts are labor-intensive and conflate the high-level design of new AI systems with the low-level exploration of teaching LMs how to follow constraints.

We propose *LM Assertions*, a novel programming construct designed to enforce user-specified properties on LM outputs within a pipeline. Drawing inspiration from runtime assertions and program specifications in traditional programming, LM Assertions are boolean conditions that express the desired characteristics of LM outputs. Besides serving as conventional runtime monitors, LM Assertions differ from traditional programming language assertions with multiple novel *assertion-driven optimizations* to improve LM programs.

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

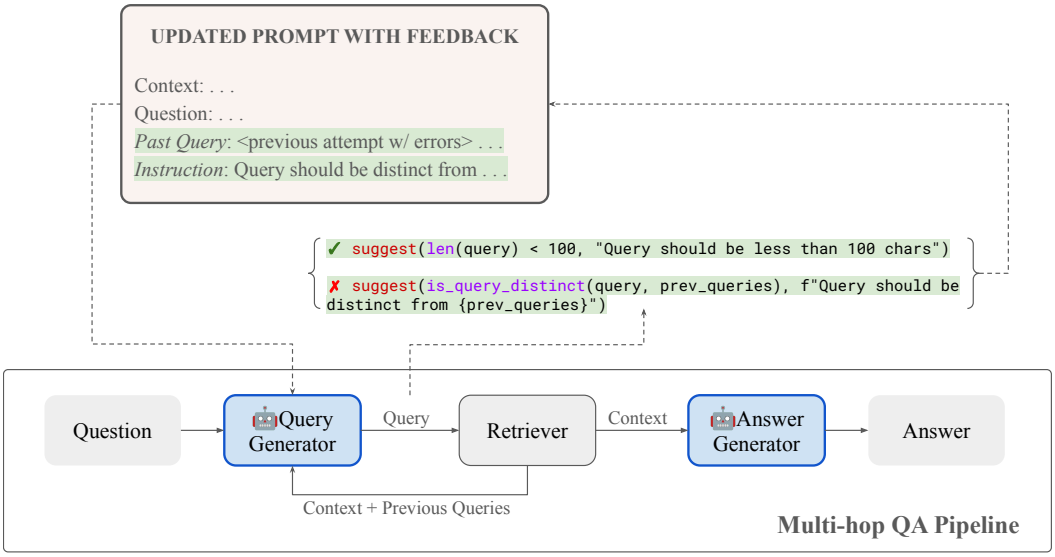

Figure 1: An LM pipeline for multi-hop question-answering tasks with a retriever. We introduce two soft Suggestions: (1) query to retriever should be less than 100 characters; (2) query to retriever should differ from previous queries. For instance, if the second suggestion fails, LM Assertion constructs a new prompt to retry the Query Generator module with additional fields, highlighting the previously generated query and a user-defined error message to help the LM refine its generation.

Assertion-driven backtracking. LM Assertions can facilitate runtime self-refinement in LM pipelines at inference time. When a constraint fails, we allow the pipeline to backtrack and retry the failing module. Upon the retry attempt, LM Assertions provide feedback and inject the erring outputs and user-specified error messages within the prompt, thereby guiding the LM to introspectively self-refine outputs. Figure 1 illustrates this within an LM pipeline.

Assertion-driven example selection. LM Assertions enable guided prompt optimization for in-context learning. Integrated with existing in-context learning frameworks, they help select high-quality few-shot examples that adhere to the user-specified constraints, which can teach LM programs to perform optimal behavior.

Counterexample bootstrapping. Another important contribution of LM assertions during prompt optimization and example selection is developing demonstrations that contain failed examples and traces to fix the errors. When such counterexamples are mixed with bootstrapped high-quality few-shot examples, the LM is presented with both constructive and instructive behavior and is thereby more likely to avoid the same mistakes and perform optimally than compared to prompting LMs without assertion-driven backtracking.

Both example selection and example bootstrapping are essential to in-context learning. When the dataset is labeled and aligned with the domain-specific objective, selecting high-quality examples provides LM with better context. On the other hand, finding datasets with corresponding labels is hard for many domains. Then, it is important to bootstrap demonstrations from existing inputs. While previous work [11] supports bootstrapping demonstrations, we argue that assertion-driven bootstrapping with counterexamples is more effective.

We evaluate the effectiveness of LM Assertions on four varied knowledge-intensive tasks: multihop question answering (MultiHopQA), long format question answering (LongFormQA), formatted quiz generation (QuizGen), and valid tweet generation (TweetGen). Our experiments show that LM Assertions and assertion-driven backtracking significantly improve LM programs, e.g., from generating 34.0% well-formatted quiz questions in JSON to 99.2%. Then, with assertion-driven example bootstrapping and counterexample bootstrapping, we see an increase from 0.0% to 98.0% in writing tweets without hashtags in TweetGen and a consistent boost on almost all other assertion metrics. Finally, with LM Assertions and all assertion-driven optimizations, we see a maximum gain

from 30.2% to 86.1% in generating valid quiz questions based on downstream task performance on composite evaluation metrics.

Our contributions are, first, introducing ***LM Assertions*** as a novel abstraction for language model programming. Then, we present three novel optimizations for LM pipelines: **assertion-driven backtracking** during inference to help pipelines self-refine, **assertion-driven example selection** to choose more effective few-shot examples for in-context learning, and **counterexample bootstrapping** to augment few-shot examples with erroneous results and corresponding fixes to help models become more reliable at complying to user-desired constraints.

2 Related Work

Programming LM pipelines. Chaining language model calls with retrieval models and tools has become increasingly popular for tackling complex tasks when single LM calls are insufficient. Such systems, often referred to as LM pipelines or compound AI systems [22], are showing significant performance gains. Frameworks like DSPy[11] or LangChain[4] support the development of such LM pipelines from a programmatic perspective.

An outstanding problem with LM pipelines is that when one module involving an LM call fails to generate the desired input, it causes errors that propagate and makes it less likely for succeeding LM modules to produce accurate responses and optimal performance. For example, in Figure 1, if *Query Generator* cannot output high-quality queries to the retrieval model, the *Answer Generator* will produce a response with low relevance, thereby failing to produce a correct response to the original input question. For this reason, asserting expectant behavior on the output of LMs for single components and the overall prompting pipeline is essential to producing optimal performance from compound AI systems.

Programming with constraints. Programming with constraints is standard in most programming languages. Languages like Java [1] and Python [14] support assertions as first-class statements to perform runtime checks of certain properties. However, systems mostly use runtime checks to warn the programmer or abort the execution. When used with `try {...} catch {...}` blocks, programming language assertions are used to break the execution of the original erring code and run the mitigating code within the `catch`. However, each LM component often reflects some level of non-determinism when programming LM pipelines, as each call can yield significantly different responses. As a result, aborting the execution or executing error-mitigation code when assertions are violated is less desirable. Instead, LM Assertions showcase specialized semantics that allows retrying the failing LM call and resuming execution with a potentially correct response. These semantics of retrying and resuming execution are partly inspired by continuation-passing-style compilation [19] and algebraic effects and handlers [13].

Constraints for machine learning models. Kang et al. [9] proposed a concept called model assertions, which can be used to monitor the behavior of ML models and to improve the quality of a model in training through data collection and weak supervision. LM Assertions and the pipeline optimizations applied in correspondence with the execution differ from model assertions in multiple ways: first, LM Assertions can be used for backtracking an LM pipeline to retry a failing module for self-refinement, which drastically improves the performance of the pipeline, second, LM Assertions can be used as filters to select better examples for few-shot learning; finally, LM Assertions aid in generating counterexamples and fixing traces, which further enhance the LM pipeline to learn from past failures and improve towards downstream metrics.

More recent efforts on generating controllable outputs for language models include LMQL [2], NeMo Guardrails [15], SGLang [23], etc. Although these systems permit some computation constraints, they work precisely on a single LM call without consideration of a wider prompting pipeline setting, which misses the assertion-driven optimization opportunities proposed by this work.

3 LM Pipelines: A Motivating Example

Aiden is a developer building an LM pipeline for multi-hop question-answering. The task involves the LM performing a series of inferential steps (multi-hop reasoning) before answering a question while utilizing a retriever to get relevant context.

Aiden may design the pipeline in Figure 1, where the LM generates search queries to collect relevant context from a retriever for iterations and aggregate them to generate the answer. However, many issues with the pipeline might affect its performance. For instance, since questions are complex, the generated search query could be long and imprecise, resulting in irrelevant retrieved context. Another issue is that similar multi-hop queries would result in redundant retrieved context. One might observe that these are properties of generated queries that are *computationally verifiable* and, if expressible as *constraints* on the pipeline, can improve its performance.

Figure 1 shows an example of using LM Assertions. To mitigate the issues above, Aiden introduces two *soft* LM Assertions (LM Assertions that are desired but not required, see Section 4 for a formal definition of *softness*): first, they restrict the length of the query to be less than 100 characters, aiming for precise information retrieval. Second, they require the query generated at each hop to be dissimilar from previous hops, discouraging retrieval of redundant information. They specify these as *soft constraints* using the `Suggest` construct. The force of this construct is to allow the pipeline to backtrack to the failing module and try again. On retrying, the LM prompt also contains its past attempts and suggestion messages, enabling constraint-guided self-refinement.

When the self-refinement attempt succeeds with responses that adhere to these *soft* constraints, the corresponding input and output are composed into a high-quality demonstration for future in-context learning. The bootstrapped demonstration satisfies all of Aiden's specifications, containing errors to avoid and detailed examples of fixing erring outputs to pass the constraints.

In Section 6, we evaluate this pipeline on the HotPotQA [20] dataset. Enabling the developer to express two simple suggestions improves the retriever's recall (by 8.0%–11.2%) and the accuracy of generated answers (by 2.4%–19.4%).

4 Language Model Assertions

We introduce **LM Assertions**, a novel programming construct that enables enforcing user-specified properties directly on LM outputs within a prompting pipeline. Drawing inspiration from the principles of runtime assertions in programming, LM Assertions are boolean conditions that articulate the desired characteristics of LM outputs, thereby bridging the deterministic world of programming and the probabilistic nature of language models. Below, we summarize our key design choices for this programming construct.

Precise and natural to programming. Consider a general setup of a language model $\mathcal{L}$ that generates output $o \in \mathcal{O}$ based on a given input $i \in \mathcal{I}$, where $\mathcal{I}$ and $\mathcal{O}$ denote the spaces of all possible inputs and outputs, respectively. An LM Assertion $\mathcal{A}$ is a predicate over $\mathcal{O}$, such that $\mathcal{A} : \mathcal{O} \rightarrow \{true, false\}$, indicating whether a given output satisfies the specified constraint:

$$\mathcal{A}(o) = \begin{cases} true & \text{if } o \text{ satisfies the constraint} \\ false & \text{otherwise} \end{cases}$$

$\mathcal{A}$ can be arbitrary code that checks for user-specific constraints, naturally extending traditional assertions in programming to language models. Formulating this as a programming construct makes it *precise*, offering developers direct control over LM outputs.

Iteration with feedback. In the traditional (static) setting described above, the LM (and overall pipeline) is not set up to refine its outputs. This is becoming increasingly important as we move from static pipelines to agentic workflows with language models. We enable this by simply extending assertions with *retry* semantics. Our key observation is that assertions, on failure, can also provide concrete *feedback* for the LM to refine its outputs via accompanying *assertion messages* m.

Let o_0 be the LM's initial response on an initial input i_0 (i.e., $\mathcal{L}(i_0) = o_0$). If $\mathcal{A}(o_0) = false$, indicating that the output does not satisfy the constraint, the pipeline enters a retry state, attempting to

generate a new output that adheres to $\mathcal{A}$. On retry, the erroneous response o_0 and the accompanying assertion messages m are written into the initial input i_0 as i_1. The assertion message and failed response serve as feedback to the LM to avoid outputs in $\mathcal{O}$ that can cause the assertion to fail. Then, the LM is called with new input i_1 to produce $\mathcal{L}(i_1) = o_1$. This process can be formalized as a sequence of attempts $\{o_0, o_1, \ldots, o_n\}$ until $\mathcal{A}(o_k) = true$ or a maximum number of retries n is reached ($k \geq n$).

Flexibility in strictness. To make LM Assertions flexibly applicable in a wide range of domains and tasks, we delineate the general idea into two concrete programming constructs: hard Assertions $\mathcal{A}_h$ and soft Suggestions $\mathcal{A}_s$. With a maximum number of allowed retries as n (could be user-defined), we can define $\mathcal{A}_h$ and $\mathcal{A}_s$ as following:

1. **Hard Assertions** ($\mathcal{A}_h$): Denoted by the syntax `Assert`, if $\mathcal{A}_h(o_k) = false$ for all $k \leq n$, the process is terminated, indicating a critical failure to meet a constraint.
2. **Soft Suggestions** ($\mathcal{A}_s$): Denoted by the syntax `Suggest`, if $\mathcal{A}_s(o_k) = false$ for all $k \leq n$, the construct logs a warning but continues, reflecting a non-critical deviation.

Composability of assertions. Lastly, LM Assertions are composable in that a sequence of assertions can be applied in a user-specified order to refine the output progressively toward the desired state. This composability allows developers to construct complex constraint specifications by combining simpler assertions to enforce multifaceted requirements on LM outputs. For instance, LM Assertions for factuality ($\mathcal{A}_{fact}$), format adherence ($\mathcal{A}_{format}$), and grammatical correctness ($\mathcal{A}_{gram}$) applied sequentially can ensure a generated technical report meets all requirements.

5 Assertion-Driven Optimizations

Section 4 highlights how LM Assertions extend beyond simple guardrails but unlock new potential in LM prompting pipelines. Applying assertions with *retry* semantics to a module in an LM pipeline can drastically improve the performance of overall downstream tasks. This allows the application of assertions as constraints in an optimization problem over prompts toward a desired output, which is notably essential in the case of in-context learning (ICL)–a *paradigm that allows language models to learn tasks given only a few examples through demonstrations* [3]. Many studies have shown that the performance of ICL strongly relies on the quality of demonstrations selected [5]. We expand on the use case of assertions as constraints to prompt optimizers for ICL.

5.1 Assertion-Driven Backtracking

Both `Assert` and `Suggest` allow the pipeline to backtrack to a failing LM call and self-refine its outputs with the *retry* mechanism described in Section 4. When the pipeline contains multiple LM modules, LM Assertions enable backtracking to arbitrary modules that produce an undesirable response at *any* time. In the LM pipeline and agentic settings, backtracking and fixing erroneous responses of a single LM component enables performance optimization for subsequent downstream modules and the overall task objective.

To keep the programming simple, adding LM assertions automatically instruments the LM pipeline for retrying and optimizations. The user interface for LM Assertion hence draws similarities to standard programming assertions (e.g., Python's `assert`); however, during inference, the instrumented retrying mechanism can alter the control flow of the pipeline and perform assertion-driven backtracking. In Appendix A, we describe the implementation of these constructs and the instrumentation.

5.2 Assertion-Driven Example Selection

LM Assertions are helpful for in-context learning, particularly in selecting high-quality few-shot examples as demonstrations [5] to enhance performance. LM pipelines consist of several modules (each performing a subtask) with specialized inputs. When programming such systems for ICL tasks, the demonstrations must also be specialized for each sub-module to ensure optimal performance within each pipeline component, further optimizing the overall pipeline performance.

For example, in our study on quiz generation (Section 6), the LM generating quiz choices is expected to produce the output in valid JSON format. To effectively perform ICL for this pipeline, an LM assertion $\mathcal{A}$ can check format validity and only select demonstrations (i_k, o_k) that pass the assertion $\mathcal{A}(o_k) = true$. This bootstraps a set of "valid" few-shot examples for ICL and improves the accuracy of the overall LM pipeline. Our experiments suggest that LM Assertions are quick and intuitive one-liners that substantially improve the performance of example selection for ICL, especially when such labels are not present within the dataset (which is generally the case for intermediate steps in a pipeline) or do not meet specialized requirements of domain-specific tasks.

5.3 Assertion-driven Counterexample Bootstrapping

An exciting and novel outcome of LM assertions for ICL is that we can go beyond collecting positive few-shot examples. As in Section 4, the trajectory of a language model $\mathcal{L}$ are input-output pairs $\{(i_0, o_0), (i_1, o_1), \ldots, (i_k, o_k)\}$ that define the sequence of retry attempts taken. Here, given $\mathcal{A}(o_k) = true$, all previous attempts are failures of the assertion. We thus augment the demonstration (i_k, o_k) to be a counterexample demonstration with the previous erroneous response o_{k-1}, instructions on how to fix it from the assertion message, and the corrected response o_k. Counterexamples are, therefore, negative demonstrations to avoid language models making similar mistakes. In addition, the demonstrations of the LM *fixing* assertion failures improve the LM's ability to adhere to constraints.

Notably, assertion-driven examples and counterexample bootstrapping show promising results for ICL, even without inference-time retrying. This suggests that LM Assertions can be compiled into useful static hints for in-context learning without incurring any runtime overhead for LM pipelines.

6 Evaluation

6.1 Tasks & Metrics

We study various aspects of LM Assertions on 4 interesting variants of the popular HotPotQA [20] task. These tasks represent real-world use cases of LM pipelines where multiple calls to LMs and other retrieve models are necessary:

- **T1** *MultiHopQA*: A complex question-answering task involving generating multi-hop search queries for questions and using the retrieved context to generate the correct answer.
- **T2** *LongFormQA*: A more demanding question-answering task, where the generated answer must contain citations that are faithful to the retrieved context information.
- **T3** *TweetGen*: A variant of HotPotQA, where the generated answer is expected to be a concise and engaging "tweet" that is also faithful to the retrieved context.
- **T4** *QuizGen*: A task involving generating candidate answer choices for HotPotQA questions in a `JSON` format, with distractor choices and the correct answer.

We evaluate each task with two metric categories:

- **Intrinsic Quality** measures the degree to which the outputs conform to the LM Assertions specified within the program. This metric is a benchmark for the system's ability to pass internal validation checks and user specifications.
- **Extrinsic Quality** measures how LM Assertions affect downstream performance, often on task-specific properties we cannot assert directly without access to ground-truth labels. Here, assertions provide guidance that indirectly influences overall performance.

These two metrics respectively enable us to investigate the hypotheses that LM Assertions can facilitate self-correction and refinement in LM pipelines (**H1**) and that such guided self-refinement can enhance the performance of downstream applications (**H2**). We provide a more comprehensive overview of the advanced tasks LongFormQA (Appendix B.1), QuizGen (Appendix B.2) and TweetGen (Appendix B.3), evaluated metrics, and applied constraints in Appendix B.

6.2 Dataset and Models

We utilize the HotPotQA [20] dataset for each task in the open-domain "fullwiki" setting. We then partition the official training set into subsets: 70% for training and 30% for validation. We only focus

Strategy	In-context Learning	Assertion	
		Runtime	Example Bootstrapping & Selection
`Vanilla`	✗	✗	—
`Compile`	✓	✗	✗
`Infer w/ Assert`	✗	✓	—
`Compile w/ Assert`	✓	✗	✓
`C+Infer w/ Assert`	✓	✓	✓

Table 1: Summary of assertion enabled strategies Section 6.3. `Vanilla` is the baseline pipeline with inference only, and `Compile` is the baseline with naive in-context learning for few-shot prompts. `Infer w/ Assert` supports assertion-driven backtracking for inference only, `Compile w/ Assert` incorporates assertion-driven example selection and counterexample bootstrapping during compilation. Finally, `C+Infer w/ Assert` contains all the assertion-driven optimizations during in-context learning and inference.

on examples labeled as "hard" within the dataset to align with the criteria marked by the official validation and test sets. For training and development sets, we sample 300 examples each, and for testing, we sample 500 examples.

We use the official Wikipedia 2017 "abstracts" dump of HotPotQA using a ColBERTv2 [16] retriever for retrieval. We test the program using OpenAI's `gpt-3.5-turbo` [3] with `max_tokens=500` and `temperature=0.7` for our experimental setup.

6.3 Strategies

Table 1 summarizes the five strategies in which LM Assertions (particularly `Suggest`) can be utilized for each task. First, we distinguish *uncompiled* strategies (i.e., zero-shot), `Vanilla` and `Infer w/ Assert`, that predict responses directly from raw input and *compiled* strategies (i.e., few-shot), `Compile`, `Compile w/ Assert`, and, `C+Infer w/ Assert`, that contain demonstrations compiled by a prompt optimizer. Here. we use the DSPy compiler [11] and its `BootstrapFewShotWithRandomSearch` optimizer. This optimizer implements an in-context learning algorithm that bootstraps and searches over a training dataset to curate demonstrations for the LM pipeline optimized for a user-specified metric. Once this *compilation* of few-shot examples is completed, the optimized program is used for inference, leading to enhanced downstream task performance.

Baselines include the `Vanilla` strategy that performs zero-shot prediction with no LM Assertions applied and the `Compile` strategy that compiles few-shot in-context learning with the naive DSPy optimizer. Our strategies explore *when* assertions are applied: solely during inference (`Infer w/ Assert`) in a zero-shot setting, where assertions enable self-correction; 2) only during compilation for in-context learning (`Compile w/ Assert`), where assertions guide the selection of optimal few-shot optimizations; and 3) during both program compilation and inference (`C+Infer w/ Assert`) where assertions are used not only for selecting optimal few-shot demonstrations but also as a overlaying validation for refinement during inference and enhanced downstream performance.

For a fair comparison, we ensure `Vanilla` and `Compile` baselines know the constraints to follow by adding instructions to the prompt that describe the equivalent LM Assertion. We also report experiments without these instructions in Appendix C, where baselines perform considerably worse.

6.4 Results

Our evaluation aims to answer the following hypotheses:

- **H1** LM Assertions facilitate automated self-correction and refinement through assertion-driven backtracking for arbitrary LM pipelines by showing the LM past outputs and error messages.
- **H2** Assertion-driven backtracking with LM Assertions can also enable LM pipelines to improve downstream application performance.
- **H3** When used with compilation and prompt optimization, LM Assertions bootstrap more robust and effective examples/counterexamples, aiding the goal of complying more with the computational constraints and achieving higher downstream performance.

MultiHopQA (Dev / Test)			
Strategy	**Suggestions Passed**	**Retrieval Recall**	**Answer Correctness**
`Vanilla`	66.7 / 68.2	37.3 / 37.6	45.7 / 41.0
`Infer w/ Assert`	87.7 / 88.4	40.3 / 40.6	46.7 / 42.0
`Compile`	68.7 / 67.8	42.7 / 39.4	46.3 / 39.2
`Compile w/ Assert`	89.3 / 85.6	**45.7** / 40.2	47.3 / 39.4
`C+Infer w/ Assert`	**96.3 / 92.2**	44.0 / **43.8**	**51.0 / 46.8**

LongFormQA (Dev / Test)				
Strategy	**Citation Faithfulness**	**Citation Recall**	**Citation Precision**	**Has Answer**
`Vanilla`	77.0 / 75.4	52.3 / 51.8	58.1 / 57.4	67.7 / 60.4
`Infer w/ Assert`	84.0 / 81.4	**58.0 / 57.5**	64.2 / 62.1	67.3 / 60.8
`Compile`	79.3 / 79.6	55.7 / 52.4	66.6 / 64.1	**70.7** / 55.6
`Compile w/ Assert`	84.0 / 83.0	42.0 / 42.3	72.6 / **73.6**	67.3 / 57.8
`C+Infer w/ Assert`	**89.0 / 87.8**	44.0 / 43.5	**74.8 / 73.6**	66.0 / **58.0**

QuizGen (Dev / Test)				
Strategy	**Correct `JSON`**	**Has Answer**	**Plausible Distractors**	**Validity**
`Vanilla`	41.3 / 36.2	39.7 / 34.0	64.7 / 62.4	36.3 / 30.2
`Infer w/ Assert`	99.7 / 99.2	91.0 / 89.8	73.0 / 66.2	83.2 / 80.5
`Compile`	**100.0 / 100.0**	96.7 / 92.8	69.0 / 64.0	86.4 / 81.7
`Compile w/ Assert`	**100.0 / 100.0**	**97.3** / 94.6	69.3 / 64.4	87.4 / 83.6
`C+Infer w/ Assert`	**100.0 / 100.0**	**97.3 / 94.8**	**81.0 / 70.8**	**91.1 / 86.1**

TweetGen (Dev / Test)						
Strategy	**No "#"**	**Has Answer**	**Concise**	**Engaging**	**Faithful**	**Quality**
`Vanilla`	23.3 / 21.0	51.3 / 46.8	**99.7** / 99.6	31.0 / 33.2	**78.7 / 79.2**	34.8 / 31.3
`Infer w/ Assert`	69.3 / 66.0	50.7 / 45.0	98.7 / 99.0	36.7 / 38.0	67.7 / 71.6	39.6 / 35.3
`Compile`	0.0 / 0.0	57.0 / 48.6	**99.7 / 100.0**	31.3 / 34.4	75.7 / 77.4	35.8 / 31.4
`Compile w/ Assert`	77.3 / 76.0	55.0 / 47.8	98.7 / 98.4	66.7 / 69.8	74.7 / 73.8	47.1 / 40.1
`C+Infer w/ Assert`	**97.3 / 98.0**	**58.3 / 49.0**	99.0 / 98.2	**84.3 / 82.2**	74.7 / 75.6	**54.1 / 44.9**

Figure 2: Evaluation of each task on the validation set (Dev) and the test set (Test). Tasks are described in Section 6.1, and LM pipeline configuration are described in Table 1. For each task, we use the same LM pipeline program except for the LM Assertions. Extrinsic metrics (downstream application performance) are highlighted in grey. For each metric, higher is always better. The highest value in each column is in **bold**.

6.4.1 H1: Self-Correction via LM Assertions

To study this hypothesis, we mainly look at the *intrinsic* metrics of the tasks, i.e., metrics that check if the LM pipeline conforms to the constraints of the LM assertions introduced. In Figure 2, we observe that LM Assertions consistently provide gains for all tasks when comparing the `Vanilla` and `Infer w/ Assert` strategies. That is, in a zero-shot setting, introducing our self-refinement-based LM assertions substantially improves the pipeline's ability to conform to specs, e.g. in the MultiHopQA task (Figure 1), the number of **Suggestions Passed** increases by 29.6% for the test set.

The increase is more prominent in the QuizGen task, where the LM pipeline must generate a multiple-choice quiz question in `JSON` format. Without LM Assertions, the model pipeline struggles to generate quizzes in valid JSON (**Correct `JSON`**). However, after including constraints that the response should be in JSON and include the correct answer as one of the choices, together with backtracking and self-refinement to fix these constraints, the final answers have correct formatting 99.2% of the time and have the right answer 89.8% of the time.

6.4.2 H2: Performance via Self-Correction

Next, we focus on whether defining suggestions in the program can help achieve better downstream performance by comparing `Infer w/ Assert` with `Vanilla`. We observe that on most tasks–MultiHopQA, LongFormQA, and QuizGen–we get a moderate to large improvement on extrinsic/downstream metrics (grey columns) when suggestions are defined. Notably, in QuizGen, the quiz's overall **Validity** generated increases from 30.2%

However, on tasks like TweetGen, we do not see a significant increase in the overall **Quality** of the generated tweet on the test set. We believe this is a case of "conflicting suggestions", where sequentially defined suggestions can override each other's impact if they are hard to disentangle during self-refinement. We observe similar behavior in a few experiments in the compiled strategies of `Compile w/ Assert` and `C+Infer w/ Assert` and display a few examples in Appendix D.

6.4.3 H3: Compiling with LM Assertions

Then, we explore an exciting use case of LM Assertions to serve as the filter and optimizer for few-shot demonstrations in prompt optimization. We evaluate all four tasks on three settings: the baseline `Compile`, where the program utilizes a DSPy optimizer to bootstrap few-shot examples and perform in-context learning with selected examples; `Compile w/ Assert`, where we enable suggestions in the bootstrapping and example selection process only; and finally, `C+Infer w/ Assert`, where suggestions and self-refinement are enabled in both in-context learning phase and the compiled program during inference.

By comparing `Compile` with `Compile w/ Assert`, we find that constructing few-shot examples that adhere to LM Assertions and show the self-refinement traces in the demonstrations makes the LM pipeline more likely to adhere to the same guidelines, even without runtime self-correction and backtracking. For example, in the TweetGen experiment, the strategy compiled with suggestions has 69.8% **Engaging** tweets, while the baseline few-shot strategy only generates 34.4%. Overall, compiling with suggestions helps tweet generation gain 43.0% more overall **Quality**. For other tasks, too, compiling with assertions almost always shows stronger performance in intrinsic and extrinsic metrics.

A surprising finding for TweetGen is the decrease in engagement (**Engaging**) when compiling with assertions. We inspect the responses of `Compile w/ Assert` and find that the tweets are short, thus less engaging. We suspect the following reasons: first, the user-provided instruction to fix this suggestion may not be precise enough for an LLM to follow. Second, as we mentioned in the analysis for **H2**, some LM Assertions might conflict with each other, making discrete optimization of prompts challenging to satisfy all constraints.

Finally, we put everything together and build `C+Infer w/ Assert` where suggestions are enabled *at all times*. This setting performs best for most intrinsic metrics over all other strategies due to the high-quality few-shot examples collected and runtime self-refinement. In the MultiHopQA question answering task, the compiled module with suggestions produces 14.2% more correct answers than the zero-shot baseline. In QuizGen, the zero-shot baseline only generates 30.2% valid quiz questions, while the final compiled program is valid 86.1% of the time. Similarly, in TweetGen, we see a 43.45% increase in quality tweets. In LongFormQA cited long passage question answering, although all the suggestions are more likely to pass, the answer inclusion (**Has Answer**) metric slightly dropped; this suggests the opportunities to find better LM Assertions for this program that can potentially influence the downstream tasks.

7 Conclusion

We introduced LM Assertions, a construct to express arbitrary computational constraints on the behavior of LMs. LM Assertions provide a new way to program LM pipelines that subsume self-refinement, backtracking, and constraint-aware optimization with a single intuitive programming construct. We discuss several optimizations and directions LM Assertions unlock for in-context learning to achieve high performance on complex pipelined tasks. Our evaluations show substantial gains on four case studies, reporting both intrinsic (i.e., assertion-specific) and extrinsic (i.e., downstream) task metrics. Overall, we hope to open avenues for programming more reliable applications of language models with intuitive constructs such as assertions.

References

[1] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - java with assertions. In K. Havelund and G. Rosu, editors, *Workshop on Runtime Verification, RV 2001, in connection with CAV 2001, Paris, France, July 23, 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 103–117. Elsevier, 2001.

[2] L. Beurer-Kellner, M. Fischer, and M. Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, 2023.

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

[4] H. Chase. LangChain, Oct. 2022.

[5] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, and Z. Sui. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.

[6] K. D'Oosterlinck, O. Khattab, F. Remy, T. Demeester, C. Develder, and C. Potts. In-context learning for extreme multi-label classification. *arXiv preprint arXiv:2401.12178*, 2024.

[7] C. Hokamp and Q. Liu. Lexically constrained decoding for sequence generation using grid beam search. *arXiv preprint arXiv:1704.07138*, 2017.

[8] J. E. Hu, H. Khayrallah, R. Culkin, P. Xia, T. Chen, M. Post, and B. Van Durme. Improved lexically constrained decoding for translation and monolingual rewriting. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 839–850, 2019.

[9] D. Kang, D. Raghavan, P. Bailis, and M. Zaharia. Model assertions for monitoring and improving ml models. *Proceedings of Machine Learning and Systems*, 2:481–496, 2020.

[10] O. Khattab, K. Santhanam, X. L. Li, D. Hall, P. Liang, C. Potts, and M. Zaharia. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive nlp, 2022.

[11] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts. Dspy: Compiling declarative language model calls into self-improving pipelines. *ICLR*, 2024.

[12] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, et al. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

[13] G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, Volume 9, Issue 4, Dec. 2013.

[14] Python Software Foundation. 7. simple statements. `https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement`, 2023. Accessed: 2023-12-01.

[15] T. Rebedea, R. Dinu, M. Sreedhar, C. Parisien, and J. Cohen. Nemo guardrails: A toolkit for controllable and safe llm applications with programmable rails, 2023.

[16] K. Santhanam, O. Khattab, J. Saad-Falcon, C. Potts, and M. Zaharia. Colbertv2: Effective and efficient retrieval via lightweight late interaction. *arXiv preprint arXiv:2112.01488*, 2021.

[17] Y. Shao, Y. Jiang, T. A. Kanell, P. Xu, O. Khattab, and M. S. Lam. Assisting in writing wikipedia-like articles from scratch with large language models, 2024.

[18] N. Shinn, F. Cassano, A. Gopinath, K. R. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[19] G. L. Steele. Rabbit: A compiler for scheme. Technical report, USA, 1978.

[20] Z. Yang, P. Qi, S. Zhang, Y. Bengio, W. W. Cohen, R. Salakhutdinov, and C. D. Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*, 2018.

[21] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023.

[22] M. Zaharia, O. Khattab, L. Chen, J. Q. Davis, H. Miller, C. Potts, J. Zou, M. Carbin, J. Frankle, N. Rao, and A. Ghodsi. The shift from models to compound ai systems. `https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/`, 2024.

[23] L. Zheng, L. Yin, Z. Xie, J. Huang, C. Sun, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng. Efficiently programming large language models using sglang, 2023.

A Implementation

We introduce the proposed LM Assertions as plug-in interfaces in the DSPy framework according to the semantics in Section 4. Next, we describe details about the design of our APIs and how we implement the semantics of both `Assert` and `Suggest` in DSPy.

A.1 API Design

```
dspy.Assert(constraint: bool, msg: Optional[str], backtrack: Optional[module])

dspy.Suggest(constraint: bool, msg: Optional[str], backtrack: Optional[module])
```

We inherit a simple API design for LM Assertions. Both suggestions and assertions take a boolean value `constraint` as input. Note that the computation for `constraint` can invoke other DSPy modules, potentially calling the LM to inform the result for the constraint. Then, the user provides an optional error message, which is used for error logging and feedback construction for backtracking and refinement. Finally, to enable backtracking, both `dspy.Assert` and `dspy.Suggest` contains an optional `backtrack` argument, which points to the target module to backtrack to if the constraints fail.

A.2 Error Handlers

To implement various strategies of both assertions and suggestions for different use cases, we exploit Python's native error and exception handling.

We encode error-handling logic as function wrappers. To that extent, we provide a primitive `constraint_tranform` to wrap any DSPy module with handlers. When the constraints in `dspy.Assert` and `dspy.Suggest` are false, they raise `AssertionError` and `SuggestionError`, respectively. Then, the dedicated error handling clause in the function wrapper can reroute the errors to the correct semantics.

As a result, the program's behavior after an assertion or suggestion error is completely controlled by the handlers used. To support flexibility in using LM Assertions with DSPy, we implement several composable handlers, such as disabling suggestions and assertions, suppressing assertion errors with logging, etc.

The default handlers follow the semantics as described in Section 4 to enable self-refinement. That is, we allow R retry attempts for `AssertionError` and `SuggestionError` by backtracking to the failing LM. After R retry attempts, an `AssertionError` will be raised while `SuggestionError` will only be logged silently.

A.3 Backtracking

To implement backtracking in DSPy, we introduce a new auxiliary *meta*-module called `Retry`. This module is a lightweight wrapper for any DSPy module, providing additional information about all previously unsuccessful predictions. When DSPy determines the need to backtrack to a specific module, it calls `Retry`. As shown in Figure 1, the `Retry` module automatically adds the failed predictions and the corresponding user-defined error messages raised to the prompt. Then, the LM pipeline can backtrack to the previously failed module with this updated prompt. In this way, the original module to refine is self-aware and informed of past attempts and errors on them. Consequently, this empowers the LM to develop more informed and error-avoiding generations in subsequent iterations of self-refinement.

B Case Studies

B.1 LongFormQA

B.1.1 Task

In this task, we build on the Multi-Hop QA (Figure 1) task by expecting long-form answers to questions that include citations to referenced context.

```python
class LongFormQAWithAssertions(dspy.Module):
  def __init__(self, passages_per_hop=3):
    self.retrieve = dspy.Retrieve(k=passages_per_hop)
    self.generate_query = dspy.ChainOfThought("context, question -> query")
    self.generate_cited_paragraph = dspy.ChainOfThought("context, question -> paragraph") #has
      field description to include citations

  def forward(self, question):
    context = []

    for hop in range(2):
      query = self.generate_query(context=context, question=question).query
      context += self.retrieve(query).passages

    pred = self.generate_cited_paragraph(context=context, question=question)
    dspy.Suggest(citations_check(pred.paragraph), "Every 1-2 sentences should have citations:
     'text... [x].'")

    for line, citation in get_lines_and_citations(pred, context):
      dspy.Suggest(is_faithful(line, citation), f"Your output should be based on the context:
     '{citations}'.")

    return pred
```

Figure 3: DSPy program with LM Assertions for long-form paragraph multi-hop question answering task with a retriever. We introduce two suggestions: (1) asserting every 1-2 sentences has a citation; (2) every text segment preceding a citation is faithful to its cited reference.

Figure 3 shows an implementation of this task in DSPy. As shown, it is nearly identical to Figure 1 outside of the introduction of a new `dspy.ChainOfThought` module that generates cited paragraphs referencing the retrieved context. With this task and LM pipeline, we aim not just to produce accurate answers but to generate well-structured long-form answers that are faithful to the retrieved context.

B.1.2 Metrics

We assess intrinsic performance using a sophisticated metric, Citation Faithfulness. In this metric, a small DSPy program uses the LM to check if the text preceding each citation appropriately supports the cited context. Our check outputs a boolean for faithfulness, which is then averaged across the citations in the output to aggregate a metric for evaluation. As extrinsic metrics, we use: (1) Answer Correctness, verifying if the `gold` answer is correctly incorporated; (2) Citation Precision, gauging the proportion of correctly cited titles; and (3) Citation Recall, measuring the coverage of `gold` titles cited.

B.1.3 Constraints Specified

As a simple initial check, we include a `Suggest` statement that requires every 1–2 of sentences generated has citations in an intended format. This is checked by a simple Python function `citations_check`. As a more sophisticated check, we `Suggest` that the text preceding any citation must be faithful to the cited context, ensuring that the reference text accurately represents the content of the cited information. Since this is a fuzzy condition, we employ a small DSPy program (one that uses the LM) to perform this check. Notably, the robust API design of `Suggest` allows the user to specify arbitrary expressions as conditional checks, such as an LM call. The goal of this `Suggest` statement is to ensure that all sentences are appropriately attributed to correct supporting sources.

B.2 QuizGen

B.2.1 Task

We introduce a new task stemming from the HotPotQA dataset in turning questions from the dataset into quiz questions by generating possible answer choices for the question in a JSON format.

```
class QuizChoiceGenerationWithAssertions(dspy.Module):
    def __init__(self):
        super().__init__()
        self.generate_choices = dspy.ChainOfThought("question, correct_answer,
    number_of_choices -> answer_choices") #has specified instruction to guide inputs ->
    outputs

    def forward(self, question, answer):
        choice_string = self.generate_choices(question=question, correct_answer=answer,
    number_of_choices=number_of_choices).answer_choices

        dspy.Suggest(format_checker(choice_string), "The format of the answer choices should
    be in JSON format. Please revise accordingly.")

        dspy.Suggest(is_correct_answer_included(answer, choice_string), "The answer choices do
     not include the correct answer to the question. Please revise accordingly.")

        plausibility_question = "Are the distractors in the answer choices plausible and not
    easily identifiable as incorrect?"

        plausibility_assessment = dspy.Predict("question, answer_choices, assessment_question
    -> assessment_answer")(question=question, answer_choices=choice_string,
    assessment_question=plausibility_question)

        dspy.Suggest(is_plausibility_yes(plausibility_assessment.assessment_answer), "The
    answer choices are not plausible distractors or are too easily identifiable as incorrect.
     Please revise to provide more challenging and plausible distractors.")

        return dspy.Prediction(choices = choice_string)
```

Figure 4: DSPy program with LM Assertions for quiz question choice generation. We introduce 3 suggestions: (1) asserting JSON format; (2) correct answer is included; (3) plausible distractor choices are present.

This task is represented by a very simple program in DSPy with a `dspy.ChainOfThought` module that generates a set of answer choices based on a defined question-answer pair and a specified number of choices. To ensure well-defined quiz questions, we aim for this task to adhere to consistent formatting and offer a set of plausible distractor answer choices alongside the actual correct answer to the question.

B.2.2 Metrics

We assess the task's intrinsic performance across the following metrics: (1) Valid Formatting; (2) Correct Answer Inclusion; and (3) Choices' Plausibility.

We verify consistent formatting by parsing the generated answer choices and checking their consistency to maintain JSON formatting of key-value pairs.

We similarly ensure that the outputted answer choices include the correct answer corresponding to the respective question from the HotPotQA dataset.

For determining the plausibility of the distractor choices, we build a DSPy program that relies on the LM to assess the quality of the answer choice questions. This relies on the inputs: question, generated answer choices, and the assessment question we provide: Are the distractors in the answer choices plausible and not easily identifiable as incorrect? This plausibility verification then outputs an assessment answer of whether the distractors are plausible or not.

For the extrinsic metric, we define a composite scoring metric that considers the intrinsic metrics above. The metric imposes that the conditions of valid formatting and correct answer inclusion are met, thereby ensuring valid quiz questions. When this case is met for the generated answer choices,

we return an average score over all three of the intrinsic metrics. If either of these conditions is not met, the score defaults to 0.

B.2.3 Constraints Specified

For the simple check of Valid Formatting, we include a `Suggest` statement that requires the format of the answer choices to be in JSON format. This is checked by a simple Python function `format_checker`.

Similarly, we verify Correct Answer Inclusion with the `Suggest` statement that indicates if the answer choices do not include the correct answer. This is checked by a simple Python function `is_correct_answer_included`.

To verify the plausibility of the answer choices to reflect strong distractor choices alongside the correct choice, we employ the `Suggest` statement to indicate if the answer choices are not plausible distractors or are too easily identifiable as incorrect. With a DSPy program in place to assess the choices, this `Suggest` statement ensures that all of the answer choices are plausible distractors.

B.3 TweetGen

B.3.1 Task

We introduce another new task derived from the HotPotQA dataset in generating tweets to answer questions.

This task mirrors the MultiHopQA task with the addition of a `dspy.ChainOfThought` module layer to utilize the retrieved context and corresponding question to generate a tweet that effectively answers the question. We aim for the task to ensure the tweet not only answers the question but is engaging to the reader and faithful to its relevant context.

B.3.2 Metrics

We assess the task's intrinsic performance across various metrics: (1) No Hashtags; (2) Correct Answer Inclusion; (3) Within Length; (4) Engaging; (5) Faithful.

We impose an intrinsic constraint to ensure none of the tweets have hashtags, ensuring all tweets maintain a consistent tweeting style.

As we do with QuizChoiceGeneration, we ensure the outputted tweet includes the correct answer corresponding to the respective question from the HotPotQA dataset.

We also ensure that the generated tweet adheres to a character count limit of 280 characters to model sample tweet behavior.

For determining the engagement of the tweet, we build a DSPy program that relies on the LM to assess this. This relies on the inputs: question, context, generated tweet, and the assessment question we provide: Does the assessed text make for a self-contained, engaging tweet? This verification outputs its assessment of whether the tweet is engaging in relation to its corresponding question and retrieved context.

We perform a similar assessment for the tweet's faithfulness, with the simple modification to the assessment question: Is the assessed text grounded in the context?

For the extrinsic metric, we define a composite scoring metric that considers all of the intrinsic metrics above. The metric imposes that the most relevant intrinsic conditions of a well-formed tweet are met, particularly if the tweet contains the correct answer to the question and is within the tweeting character limit. When this case is met for the generated answer choices, we return an average score over all five of the intrinsic metrics. If either of these conditions is not met, the score defaults to 0.

B.3.3 Constraints Specified

To verify that the tweet contains no hashtags, we include a `Suggest` statement that requires the tweet to be generated without any hashtag phrases. This is checked by a simple Python function through regex checks in `has_no_hashtags`.

```
class TweetGenerationWithAssertions(dspy.Module):
    def __init__(self):
        super().__init__()
        self.generate_tweet = dspy.ChainOfThought("question, context -> tweet") #has specified
     instruction to guide inputs -> outputs

    def forward(self, question, answer):
        context = []
        generate_query = [dspy.ChainOfThought("context, question -> query") for _ in range(2)]
        retrieve = dspy.Retrieve(k=3)
        for hop in range(2):
            query = generate_query[hop](context=context, question=question).query
            passages = retrieve(query).passages
            context = deduplicate(context + passages)
        generated_tweet = self.generate_tweet(question=question, context=context).tweet
        dspy.Suggest(has_no_hashtags(generated_tweet), f"Please revise the tweet to remove
     hashtag phrases following it.")
        dspy.Suggest(is_within_length_limit(generated_tweet, 280), f"Please ensure the tweet
     is within {280} characters.")
        dspy.Suggest(has_correct_answer(generated_tweet, answer), "The tweet does not include
     the correct answer to the question. Please revise accordingly.")
        engaging_question = "Does the assessed text make for a self-contained, engaging tweet?
      Say no if it is not engaging."
        engaging_assessment = dspy.Predict("context, assessed_text, assessment_question ->
     assessment_answer")(context=context, assessed_text=generated_tweet, assessment_question=
     engaging_question)
        dspy.Suggest(is_assessment_yes(engaging_assessment.assessment_answer), "The text is
     not engaging enough. Please revise to make it more captivating.")
        faithful_question = "Is the assessed text grounded in the context? Say no if it
     includes significant facts not in the context."
        faithful_assessment = dspy.Predict("context, assessed_text, assessment_question ->
     assessment_answer")(context='N/A', assessed_text=generated_tweet, assessment_question=
     faithful_question)
        dspy.Suggest(is_assessment_yes(faithful_assessment.assessment_answer), "The text
     contains unfaithful elements or significant facts not in the context. Please revise for
     accuracy.")
        return dspy.Prediction(generated_tweet=generated_tweet, context=context)
```

Figure 5: DSPy program with LM Assertions for tweet generation. We introduce 5 suggestions: (1) asserting no hashtags; (2) correct answer is included; (3) tweet is within character limit; (4) tweet is engaging; (5) tweet is faithful to context.

To verify the generated tweet adheres to the character limits, we impose this through the `Suggest` statement to ensure that the tweet is under the specified character limit, which we specify as 280 in our experiments. This is checked by a simple Python function `is_within_length_limit`.

Similarly, we verify Correct Answer Inclusion with the `Suggest` statement that indicates if the answer choices do not include the correct answer. This is checked by a simple Python function `has_correct_answer`.

To verify the engagement level of the generated tweet, we employ the `Suggest` statement to simply indicate whether the tweet is engaging enough as determined by the LM and DSPy program in place to assess engagement.

We conduct a similar approach for faithfulness as well, checking for the tweet's faithfulness to its retrieved context.

C Impact on Using Different LLM Instructions

We explore comparative tests in the specified instructions for the case studies mentioned above. We differentiate between a primitive instruction that aims to simply specify a task's objective and a

TweetGen w/ **Primitive Instructions** (Dev/Test)						
Strategy	**No "#"**	**Has Answer**	**Concise**	**Engaging**	**Faithful**	**Quality**
`Vanilla`	3.3 / 3.0	53.7 / **48.2**	96.3 / 97.0	35.7 / 36.4	80.0 / **81.2**	33.7 / 30.4
`Infer w/ Assert`	49.3 / 49.6	50.3 / 41.8	92.0 / 92.4	45.3 / 41.0	72.3 / 74.0	34.3 / 27.8
`Compile`	0.0 / 0.2	**55.7** / 46.2	**100 / 99.6**	47.3 / 46.6	**78.3** / 76.8	36.7 / 30.8
`Compile w/ Assert`	**98.7 / 97.4**	55.0 / 45.8	99.3 / 99.0	1.3 / 2.6	65.3 / 70.0	**40.4** / 34.3
`C+Infer w/ Assert`	41.3 / 41.0	**55.7 / 48.2**	94.7 / 93.8	**54.3 / 60.2**	76.7 / **81.2**	40.3 / **35.0**

QuizGen w/ **Primitive Instructions** (Dev/Test)				
Strategy	**Correct `JSON`**	**Has Answer**	**Citation Precision**	**Validity**
`Vanilla`	1.3 / 2.8	1.3 / 2.6	61.3 / 61.8	1.2 / 2.3
`Infer w/ Assert`	91.7 / 93.4	73.3 / 72.6	**75.0 / 69.8**	69.8 / 68.0
`Compile`	**100 / 100**	94.3 / 89.8	72.7 / 67.4	85.4 / 80.1
`Compile w/ Assert`	**100 / 100**	**95.7 / 91.4**	63.0 / 57.0	83.7 / 78.5
`C+Infer w/ Assert`	**100 / 100**	93.3 / 89.4	73.7 / 67.8	**85.8 / 81.1**

Figure 6: Evaluation of TweetGen and QuizGen task using the primitive instruction. The LM pipeline configuration are described in Table 1. For each task, we use the same LM pipeline program except for the LM Assertions. Extrinsic metrics (downstream application performance) are highlighted in grey. For each metric, higher is always better. The highest value in each column is **bold**.

complete instruction that accounts for the respective intrinsic and extrinsic metric measured for the task. These tests are conducted specifically on the TweetGen and QuizGen tasks which encompass more complex metrics. Our experiments on the complete instructions are presented in Figure 2 while we demonstrate our results on the primitive instructions below.

C.1 TweetGen

Primitive instruction: "Generate a tweet that effectively answers a question."

Complete instruction with metrics accounted for: "Generate an engaging tweet that effectively answers a question staying faithful to the context, is less than 280 characters, and has no hashtags."

C.2 QuizGen

Primitive instruction: "Generate answer choices for the specified question."

Complete instruction with metrics accounted for: "Generate answer choices in JSON format that include the correct answer and plausible distractors for the specified question."

C.3 Discussion

Based on these experiments on primitive instructions, we discovered that when the baseline pipeline only has access to high-level and generic instructions, it is almost impossible for the pipeline to follow the underlying constraints. For example, in QuizGen, the baseline `Vanilla` strategy only generates 2.8% of quizzes with **Correct `JSON`** format and 2.6% of quizzes that contains the correct answer.

However, for our assertion-driven optimization enabled pipelines, the performance on primitive experiments are still comparable to the counter-part with complex instructions. This indicates that model pipelines with LM Assertions and assertion-driven optimizations are less sensitive to instructions, requiring less effort on manual prompt tuning.

D Interesting Examples of Correlated Constraints

This section showcases selected examples that demonstrate potential conflicts between Assertions specified in the TweetGen task. Each set of examples highlights instances of two configurations that

produce outputs to achieve success on the TweetGen Metrics (Appendix B.3.2) and face the nuanced challenge of balancing these constraints in execution.

Example 1 Both results accurately include the correct answer. However, `Compile w/ Assert` does not pass the **Engaging** constraint despite the specified assertion but ensures the **No "#"** constraint is met whereas `C+Infer w/ Assert` violates the **No "#"** constraint while satisfying the **Engaging** constraint.

Question: What was the name of the treaty that made Hungary a landlocked state which contained the Kolozsvar Ghetto?

> `Compile w/ Assert`: The Treaty of Trianon made Hungary landlocked and led to the existence of the Kolozsvar Ghetto. This lesser-known ghetto was located in Kolozsvár, Kingdom of Hungary (now Cluj-Napoca, Romania).
>
> `C+Infer w/ Assert`: The Treaty of Trianon, signed in 1920, reshaped Hungaryś destiny, leading to its landlocked status and the emergence of the Kolozsvar Ghetto. Uncover the gripping tale of this pivotal moment in history and its lasting effects. #HistoryUnveiled

Answer: Treaty of Trianon

Example 2 Both results do not include the correct answer. However, both configurations produce engaging outputs. Yet `C+Infer w/ Assert` does not conform to the **No "#"** constraint while `Compile w/ Assert` does.

Question: Which American car rental company is also a member of the Association of Car Rental Industry Sytems Standards?

> `Compile w/ Assert`: "Enterprise, a leading American car rental company, is also a proud member of the Association of Car Rental Industry Systems Standards. Rent with confidence and enjoy a seamless experience with Enterprise!"
>
> `C+Infer w/ Assert`: "Looking for a car rental company that meets industry standards? Look no further than ACRISS member Enterprise! With their commitment to excellence, you can trust them for a seamless rental experience. Get ready for a smooth ride! #CarRental #Enterprise"

Answer: Budget Rent a Car

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: All the numbers and claims in the abstract and introduction are directly based on the evaluation section.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: At the end of the evaluation section, we discuss a failure mode for assertions where the user-provided assertions conflict with each other, resulting in suboptimal performance. We provide detailed examples in the appendix, and highlight how such limitations can be avoided by designing more robust LM assertions (while acknowledging this is a challenging task).

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory Assumptions and Proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [NA]

Justification: We do not make any theoretical conclusion and rely on empirical results to validate our claims.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental Result Reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: We provide clear semantics about LM Assertions and assertion-driven optimizations in addition to the majority part of the experiment code (in appendix). Our code, data, and experiment notebooks will be made public for the camera-ready paper if accepted.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in

some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: We open-sourced our code and experiment but did not link them due to anonymity. Our code, experiments, and links (beyond what is included in this paper) will be made public for the camera-ready paper if accepted. The data used is open-sourced online.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental Setting/Details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: We report the key settings of the experiment in Section 6.2. Additional details can be found in the code and instructions that will be made public for the camera-ready paper if accepted.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment Statistical Significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: Experimental results involve the evaluation of LLM outputs which does not require traditional statistical error reporting given the nature of the benchmark tasks and deterministic behavior.

   Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [No]

   Justification: The experiments in this paper are focused on benchmarking evaluation of LLM outputs and can be replicated without the need for specific compute infrastructure, making them accessible to anyone with basic computing resources that could access any LLM API.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Our research conforms to the NeurIPS Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

Justification: We discuss how LM Assertions can assist developers in defining important LM guardrails effectively within prompting pipelines, enhancing safety and reliability in user interactions. We also acknowledge potential risks with the misuse of LM assertions to bias information from LM outputs, and consider potential mitigation strategies such as highlighting the ethics involved with user-defined assertions in prompting pipelines.

Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

Answer: [NA]

Justification: Our work focuses on regulating LM outputs and helping LM Pipelines align better with the developer's intent. Our experiments deal with open-source data and do not have any safety risks while using LM assertions.

Guidelines:

- The answer NA means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: We cite and provide references to all the assets we used in our development.

Guidelines:

- The answer NA means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [No]

    Justification: Our contributions will be made public within the open-source project with detailed documentation for the camera-ready publication if the paper is accepted.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: This paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [NA]

Justification: This paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.