TYPE-COMPLIANT ADAPTATION CASCADES: ADAPTING PROGRAMMATIC LM WORKFLOWS TO DATA

Anonymous authors
Paper under double-blind review

ABSTRACT

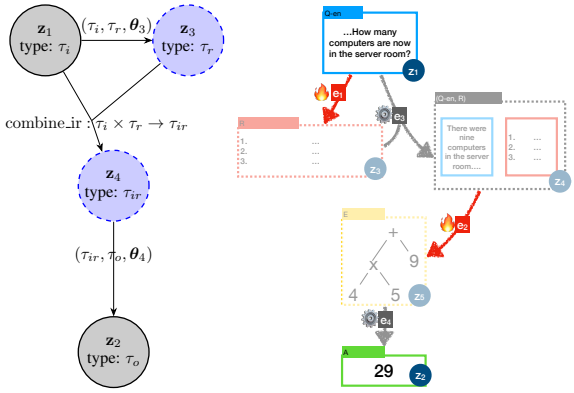
Reliably composing Large Language Models (LLMs) for complex, multi-step workflows remains a significant challenge. The dominant paradigm — optimizing discrete prompts in a pipeline — is notoriously brittle and struggles to enforce the formal compliance required for structured tasks. We introduce Type-Compliant Adaptation Cascades (TACs), a framework that recasts workflow adaptation as learning typed probabilistic programs. TACs treat the entire workflow, which is composed of parameter-efficiently adapted LLMs and deterministic logic, as an unnormalized joint distribution. This enables principled, gradient-based training even with latent intermediate structures. We provide theoretical justification for our tractable optimization objective, proving that the optimization bias vanishes as the model learns type compliance. Empirically, TACs significantly outperform state-of-the-art prompt-optimization baselines. Gains are particularly pronounced on structured tasks, improving FinQA from 12.0% to 24.7% for a Qwen 3 8B model, MGSM-SymPy from 57.1% to 75.9% for a Gemma 2 27B model, MGSM from 1.6% to 27.3%, and MuSR from 36.5% to 62.6% for a Gemma 7B model. TACs offer a robust and theoretically grounded paradigm for developing reliable, task-compliant LLM systems.

1 INTRODUCTION

Language modeling (Rosenfeld, 2018) refers to fitting a parametric probability distribution over strings (a *language model*) $p_{\boldsymbol{\theta}}$ to observed data. Large Language Models (LLMs) (Brown et al., 2020) scale both the model and training datasets to massive sizes. LLMs have an extraordinary emergent capability: once trained, these distributions can be effectively manipulated simply by *asking* — conditioning the distribution on different natural language instruction prefixes (Wei et al., 2022a) — a practice widely known as prompting.

The expressive power and accessibility of this natural language interface have catalyzed the rapid development of programmatically composed workflows and agentic systems (Khattab et al., 2022; Chase, 2022; Yao et al., 2023; Wu et al., 2024). By structuring inputs and chaining model calls, practitioners can construct complex systems capable of multi-step reasoning and interaction. However, the success of these systems is inherently subject to the pretrained LLM's capabilities in instruction following. Moreover, prompt engineering remains brittle: minor textual variations can lead to drastic performance degradation (Cao et al., 2024). This brittleness can also cause type violations in a programmatic workflow: while inference-time constrained decoding methods mitigate type violation problems, full compliance remains theoretically impossible for complex types (Lin et al., 2021) on autoregressive models. Optimizing these composed systems therefore often devolves into a difficult discrete optimization problem over the space of possible prompts — a challenge often addressed through heuristic search (Zhou et al., 2023; Pryzant et al., 2023; Yuksekgonul et al., 2025) and reinforcement learning (Jafari et al., 2024), both of which suffer from variance issues.

In this paper, we propose a return to the foundational perspective: fitting composed LLM distributions to downstream tasks as *parametric probability models*. Instead of tackling the inherent difficulties of optimizing



(a) **cot-cascade-structure** (b) **expression-cascade-structure**

Figure 1: Two TAC workflow patterns experimented in this paper. We illustrate the more complicated Fig. 1b with example node values (we also explore additional patterns in §B). Dashed-boundary nodes indicate variables whose values are not available in annotated data, and solid-boundary nodes indicate nodes with training time observable values. A main message of this work is that **we can treat an entire typed workflow as a single probabilistic program, whose parameters are lightweight PEFT modules, allowing end-to-end training with latent variables**, instead of defining workflows imperatively as fixed-parameter systems.

discrete verbal instructions, we *adapt* a composed workflow (such as ones shown in Fig. 1), as a parametric latent variable model, to maximize data likelihood. Each step in the workflow is a probabilistic typed transformation backed by a parameter-efficient fine-tuning (PEFT) adaptor, with valid typed objects as its support. Different workflows are declaratively defined as different generative stories that sequentially transform objects with either learned adaptors or deterministic algorithms. Thus, we transform the problem of workflow adaptation from an ad-hoc, discrete optimization search problem to training and inference of latent variable models. This allows us to leverage well-established machine learning techniques to optimize the entire system directly, while keeping training and inference manageable, thanks to the adaptors' parameter and computational efficiency.

This approach, which we term Type-Compliant Adaptation Cascades (TACs), is an end-to-end trainable probabilistic programming framework. As parametric latent variable models, TACs can be optimized using gradient descent methods. Moreover, as unnormalized distributions over typed objects, Posterior inference of TACs is decoupled from training, enabling techniques such as amortized inference and classification by ranking.

Our primary contributions are:

- **Framework**. We formalize typed LM workflows as probabilistic programs: each learned hyperedge is an unnormalized conditional distribution that assigns zero mass to outputs violating type contracts.
- **Theory**. We propose a tractable and theoretically-grounded training algorithm, TACSTaR. We prove that our optimization objective, while computationally efficient, correctly converges to the ideal solution as the model learns to become type-compliant. Specifically, we show that the bias in our gradient approximation vanishes as the model's adherence to type constraints increases during training (Theorems 1 and 2).
- **Practice**. Across QA, structured generation, and classification tasks that require heavy reasoning (MGSM, MGSM-SymPy, FinQA, MuSR) and model families (Gemma, Qwen), TACs consistently outperform strong DSPy prompt-optimization baselines. Gains are largest when (1) base models are smaller and (2) tasks require strict structure. For example, on MGSM-SymPy with a Gemma 27B model, TACs achieve $\mathbf{75.9}$ vs. $\mathbf{57.1}$; on FinQA, $\mathbf{34.0}$ vs. $\mathbf{12.7}$ (Gemma 27B) and $\mathbf{24.7}$ vs. $\mathbf{12.0}$ (Qwen 3 8B). With a Gemma 7B model, MGSM improves from $\mathbf{1.6}$ to $\mathbf{27.3}$, FinQA from $\mathbf{0.7}$ to $\mathbf{9.7}$, and MuSR from $\mathbf{36.5}$ to $\mathbf{62.6}$.

Summary of results. (1) Gradient-based adaptation *within* typed workflows is markedly more effective than discrete prompt search for structured tasks. (2) Flexible training- and test-time posterior inference help performance. (3) Empirically, estimated type compliance mass $\mathcal{Z}_{\boldsymbol{\theta}}$ rises rapidly during training and correlates with accuracy, supporting our theoretical justification for the unnormalized objective.

2 TYPE-COMPLIANT ADAPTOR CASCADES

The core idea of TACs is to decompose a task into a hypergraph of interconnected transformations. Formally, a TAC is represented as a directed acyclic hypergraph (DAH) $C = (\mathbf{Z}, \mathbf{E})$.[1] The acyclic constraint ensures that the workflow has a well-defined topological order for execution and guarantees termination of the generative process.

Nodes. The nodes $\mathbf{Z} = \{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_M\}$ in a TAC act as containers for typed data. Each node $\mathbf{z}_m$ is associated with a specific data type $\tau \in \mathcal{T}$, and holds string representations $\in \Sigma^*$ for τ-typed objects. Special nodes are designated as the **input node** $\mathbf{z}_1$ and the **output node** $\mathbf{z}_2$ (*e.g.*, holding the initial question of type `Q_en` and the final answer of type `A` in Fig. 1b, respectively).

Hyperedges. Hyperedges $\mathbf{E} = \{e_1, e_2, \ldots, e_K\}$ define the transformations between nodes. A hyperedge e_k connects a set of source nodes $S_k \subseteq \mathbf{Z}$ (its inputs) to a set of target nodes $T_k \subseteq \mathbf{Z}$ (its outputs). Transformations in TACs can be either learnable (LM adaptors) or fixed (deterministic algorithms):

- **LM adaptor hyperedges.** These are stochastic transformations implemented by PEFT-adapted LMs. An adaptor $(\tau_i, \tau_o, \boldsymbol{\theta})$ defines an unnormalized distribution over $\boldsymbol{y} \in \Sigma^*$ given input string $\boldsymbol{x}$:[2]
$$\tilde{p}(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta}) = p_{LM}(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta}) \mathbb{I}(\mathbf{z}_t \in \text{valid}(\tau_o)), \tag{1}$$
where $p_{LM}(\cdot \mid \boldsymbol{x}; \boldsymbol{\theta})$ is a normalized distribution over strings, conditioned on τ_i-typed string representation $\boldsymbol{x}$, and parametrized by adaptor parameters $\boldsymbol{\theta}$, and $\text{valid}(\tau_o) \subseteq \Sigma^*$ is the set of strings that represent valid τ_o-typed objects (we will further discuss them in §2.1).
- **Deterministic algorithm hyperedges.** These are fixed, non-learnable transformations, such as a self-contained Python function. A deterministic algorithm f maps an input object of type τ_i to an output object of type τ_o. Under the probabilistic view, we represent them as δ distributions:
$$\tilde{p}(\boldsymbol{y} \mid \boldsymbol{x}; f) = \delta_{\texttt{canon}(f(\texttt{parse}(x, \tau_i)))}(y) \tag{2}$$
where `canon` (see §2.1) produces a canonicalized string for an object, and `parse` converts strings back to typed objects.

2.1 INTERFACING LLMS WITH TYPED DATA: PARSING AND CANONICALIZATION

A crucial subtlety in integrating LLMs into typed workflows is bridging their native string-based operation with typed data, which is typically handled by data validation libraries such as Pydantic[3] and LangFun.[4] Here we formalize the conversion under the TAC formalism as two operations `parse` and `canon`:

[1] We use a reasoning workflow that generates domain-specific code, illustrated in Fig. 1b, as a running example. The task is to take a math question in English (input type `Q_en`), generate a step-by-step rationale (intermediate type `R`), convert the rationale into a formal arithmetic expression (intermediate type `E`), and finally, have a deterministic function evaluate this expression to produce the answer (output type `A`). This section formalizes how such an intuitive sketch is realized within the TAC framework.

[2] This distribution may be unnormalized because while p_{LM} is a distribution over all strings, Eq. (1) restricts the support to only strings that are valid instances of τ_o. Thus, the total probability mass may sum to less than 1 if the LM assigns probability to invalid strings.

[3] `https://github.com/pydantic/pydantic`

[4] `https://github.com/google/langfun`. Examples of generated prompts are listed in §N.

Parsing (`parse`). When an LM adaptor produces an output string $\boldsymbol{y}$ intended to represent an object of type τ_o, this string is validated and converted into a usable typed object by the algorithm $\texttt{parse} : \Sigma^* \times \mathcal{T} \to \mathcal{O} \cup \{\text{error}\}$.[5] For example, in Fig. 1b, $\mathbf{z}_5$ has the deterministic function e_4 as an outgoing edge. During execution of the probabilistic program, $\texttt{parse}(\mathbf{z}_5, \texttt{E})$ attempts to convert $\mathbf{z}_5$ into a SymPy expression object (typed `E`). If the conversion fails, an error is signaled. For convenience, we use $\text{valid}(\tau) = \{\texttt{parse}(\boldsymbol{y}, \tau) \neq \text{error} \mid \boldsymbol{y} \in \Sigma^*\}$ to denote valid string representations of τ.

Canonicalization (`canon`). Conversely, inputs of LM adaptor hyperedges must be converted into a consistent string format that the adaptor expects. The $\texttt{canon} : \mathcal{O} \to \Sigma^*$ operation maps a typed object to a unique string representation — we call such strings *canonicalized*. The invertibility of `canon` (*i.e.*, $\texttt{parse}(\texttt{canon}(o), \tau_o) = o$) in turn ensures that deterministic hyperedges have support over only one string given a valid input, eliminating spurious ambiguity (Cohen et al., 2012).

2.2 TACs As Programs And Distributions

TACs admit both a program view, and also a probabilistic view[6]:

- **TACs are probabilistic programs.** Executing a TAC in the forward direction involves processing data through the hypergraph, respecting the topological order of nodes and hyperedges. Using our running example from Fig. 1b: the process traverses the hypergraph, starting at the input variable $\mathbf{z}_1$ (typed `Q_en`), and ending at the output variable $\mathbf{z}_2$ (typed `A`). A general process is described in Algorithm 1.
- **TACs are also probability distributions.** TACs also define unnormalized joint probability distributions over all node assignments $\mathbf{Z}^* = (\mathbf{z}_1^*, \mathbf{z}_2^*, \ldots, \mathbf{z}_M^*)$. This score reflects the plausibility of a complete execution trace according to the model's components:

$$\log \tilde{p}_{\boldsymbol{\theta}}(\mathbf{Z}^*) = \sum_k \log \tilde{p}_{\boldsymbol{\theta}}(\{\mathbf{z}_t^*\}_{t \in T_k} \mid \{\mathbf{z}_s^*\}_{s \in S_k}; e_k), \tag{3}$$

where $\boldsymbol{\theta}$ represent all adaptor parameters used in the TAC, and $\tilde{p}_{\boldsymbol{\theta}}(\cdot|\cdot; e_k)$ is the conditional probability defined by the LM adaptor (Eq. (1)) or deterministic algorithm (Eq. (2)) associated with e_k. The unnormalized distribution view connects TACs to the broader family of language model cascades (Dohan et al., 2022), but with the key distinction that TACs are designed for end-to-end adaptation.

Estimating unnormalized marginal probabilities. LM adaptors in a TAC can be used as proposal distributions to get an importance sampling estimate of the unnormalized marginal probability. Let $\mathbf{z}_m$ be a node coming out of an LM adaptor, an N-sample estimate of the unnormalized probability that $\mathbf{z}_m$ equals c: $\tilde{p}(\mathbf{z}_m = c; \boldsymbol{\theta})$ is:

$$\hat{\tilde{p}}_{|\mathbf{z}_1}(m, c, N) = \sum_{n=1}^{N} \left[\frac{p_{LM}(\mathbf{z}_m = c; \boldsymbol{\theta})}{N \cdot p_{LM}(\mathbf{z}_m = \mathbf{z}_m^{(n)}; \boldsymbol{\theta})} \right] \tag{4}$$

where $\mathbf{z}_m^{(n)}$ is the n-th sample of $\mathbf{z}_m$ (possibly drawn using Algorithm 1). Equation (4) is an unbiased importance sampling estimate of the unnormalized probability $\tilde{p}(\mathbf{z}_m = c \mid \mathbf{z}_1; \boldsymbol{\theta})$ (since $\text{supp}(\tilde{p}) \subseteq \text{supp}(p_{LM})$). In general, $\mathbf{z}_m$ has an infinite support, making the *normalized* probability $p(\mathbf{z}_m = c \mid \mathbf{z}_1; \boldsymbol{\theta})$ intractable. In the special case that $\mathbf{z}_m$ has finite support, Eq. (4) can be used to estimate the *normalized* marginal probability

[5] We note that while primitive data types (*e.g.*, Python types `str` and `list`) appear in common workflows, `parse` can be any computable function, and can be leveraged by a practitioner to implement complex business logic. For example, one can define a Python custom type `CoherentDialog` where valid objects are strings deemed coherent by an external LLM-backed classifier, and adapt LM adaptors in a TAC to generate and work with such objects. Implementation details are further discussed in §E.

[6] These two views are also summarized in Table 1.

$\hat{p}(\mathbf{z}_m = c \mid \mathbf{z}_1; \boldsymbol{\theta}) = \frac{\hat{\tilde{p}}_{|\mathbf{z}_1}(m,c,N)}{\sum_{c'} \hat{\tilde{p}}_{|\mathbf{z}_1}(m,c',N)}$. We leverage Eq. (4) to estimate normalized output probabilities $p(\mathbf{z}_2 \mid \mathbf{z}_1; \boldsymbol{\theta})$, for ranking classification outputs in §4.3.

3 ADAPTING TACS

Since TACs generally define distributions over unobserved (latent) intermediate variables, Monte Carlo Expectation-Maximization (MC-EM) algorithms (Wei & Tanner, 1990) provide a suitable training paradigm for marginalized likelihood maximization.[7] MC-EM algorithms iteratively refine model parameters by alternating between an E-step (sampling latent variables) and an M-step (optimizing parameters based on these samples). The Self-Taught Reasoner (STaR) algorithm (Zelikman et al., 2022) is a notable instance of MC-EM. We generalize STaR to the TAC framework for workflows with arbitrarily typed inputs and outputs, resulting in the TACSTaR algorithm.

3.1 TACSTAR

The TACSTaR algorithm (Algorithm 3) employs an iterative MC-EM approach to train the parameters $\boldsymbol{\theta}$ of the type-compliant LM adaptors within a TAC C. As with the original STaR algorithm, TACSTaR alternates between E- and M-steps:

- **E-step: Sampling Latent Variables.** We first try to execute the TAC C as a probabilistic program under the `forward` algorithm (Algorithm 1). If `forward` succeeds, we have a complete assignment of values $\mathbf{Z}^* = (\mathbf{z}_1^*, \mathbf{z}_2^*, \ldots, \mathbf{z}_M^*)$ for all nodes in the TAC C. and can proceed to M-step. Otherwise, we attempt a **rationalization heuristic** step. Inspired by the original STaR algorithm which conditions on the correct answer in the second attempt, we construct a 'fallback' TAC, whose input node takes (x^*, y^*) as input, with the rest of the workflow unchanged. This essentially asks '*what intermediate steps would lead from x^* to y^*?*', analogous to the inverse rendering problem (Ritchie et al., 2023). A forward pass is then executed on this new TAC to sample $(\mathbf{z}_2, \ldots, \mathbf{z}_M)$, now conditioned on both the original input x^* and the desired output y^*. This encourages the generation of latent intermediate steps that are consistent with the correct final answer.
- **M-step: Parameter Optimization.** EM-style algorithms generally do MLE updates on samples collected in the E-step. As TACs are generally unnormalized models, proper MLE updates require computing partition function gradients. Denoting the partition function summing all possible assignments as $\mathcal{Z}_{\boldsymbol{\theta}} = \sum_{\mathbf{Z}'} \tilde{p}_{\boldsymbol{\theta}}(\mathbf{Z}')$, the gradient of the log-likelihood $\mathcal{L} = \log p(\mathbf{Z}^*)$ is:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L} = \nabla_{\boldsymbol{\theta}} \log \tilde{p}_{\boldsymbol{\theta}}(\mathbf{Z}^*) - \nabla_{\boldsymbol{\theta}} \log \mathcal{Z}_{\boldsymbol{\theta}}. \tag{5}$$

Estimation of the log partition function's gradients $\nabla_{\boldsymbol{\theta}} \log \mathcal{Z}_{\boldsymbol{\theta}}$ is typically expensive and can have high variance (Goodfellow et al., 2016). We thus drop this term, and optimize for the unnormalized log-likelihood $\mathcal{L}'(\boldsymbol{\theta}) = \log \tilde{p}_{\boldsymbol{\theta}}(\mathbf{Z}^*)$ instead.[8]

Tractable optimization via compliance. While ignoring the partition function gradient generally leads to biased gradient estimation, the TAC formalism ensures this strategy is both tractable and robust. This becomes evident as we rewrite $\mathcal{L}'(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}) + \log \mathcal{Z}_{\boldsymbol{\theta}}$: optimizing the unnormalized likelihood $L'(\boldsymbol{\theta})$ is equivalent to jointly maximizing the normalized likelihood $\mathcal{L}(\boldsymbol{\theta})$ and the model's type compliance (the partition function $\log \mathcal{Z}_{\boldsymbol{\theta}}$ is

[7] We acknowledge that another reasonable approach for training TACs is reinforcement learning, and note the connection between TACSTaR and RL in §A.

[8] **Remark on efficiency.** Since gradients of the log unnormalized probability decompose linearly as $\nabla_{\boldsymbol{\theta}} (\log \tilde{p}_{\boldsymbol{\theta}}(\mathbf{Z}^*)) = \sum_k \nabla_{\boldsymbol{\theta}} \log \tilde{p}_{\boldsymbol{\theta}}(\{\mathbf{z}_t^*\}_{t \in T_k} \mid \{\mathbf{z}_s^*\}_{s \in S_k}; e_k)$, computation of adaptors' gradients can be parallelized easily. This embarrassingly parallel structure ensures computational scalability, allowing the M-step to be efficiently distributed across available compute resources. Algorithm 2 computes $\log \tilde{p}_{\boldsymbol{\theta}}(\mathbf{Z}^*)$ and its gradients $\nabla_{\boldsymbol{\theta}} \log \tilde{p}_{\boldsymbol{\theta}}(\mathbf{Z}^*)$. These gradients are then used in a standard gradient-based optimization algorithm to update $\boldsymbol{\theta}$.

maximized at $\log \mathcal{Z}_{\boldsymbol{\theta}} = 0$ when $\boldsymbol{\theta}$ is well-specified). This approach is justified theoretically under the assumption that the adapted models can perfectly model type-valid outputs (*i.e.*, the model family is well-specified):[9]

Theorem 1. *Let $\boldsymbol{\Theta}$ be the entire parameter space and let $\boldsymbol{\Theta}' \subseteq \boldsymbol{\Theta}$ be the subset of well-specified parameters. Assume $\boldsymbol{\theta}^*$ uniquely maximizes the normalized likelihood $p_{\boldsymbol{\theta}}(\mathbf{z}_{2..M}|\mathbf{z}_1)$ and resides $\in \boldsymbol{\Theta}'$. Then, $\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \tilde{p}_{\boldsymbol{\theta}}(\mathbf{z}_{2..M}|\mathbf{z}_1) \implies \hat{\boldsymbol{\theta}} = \boldsymbol{\theta}^*$.*

Moreover, while optimizing $\mathcal{L}'(\boldsymbol{\theta})$ introduces a bias by ignoring the gradient term $\nabla_{\boldsymbol{\theta}} \log \mathcal{Z}_{\boldsymbol{\theta}}$, this bias is bounded below a constant multiplicative factor of $(1 - \mathcal{Z}_{\boldsymbol{\theta}})$ under the common assumption that $\|\nabla_{\boldsymbol{\theta}} p_{LM}(\cdot \mid \boldsymbol{x}; \boldsymbol{\theta})\|$ is uniformly bounded:

Theorem 2. *Let $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1 \dots \boldsymbol{\theta}_K\}$ be the union of a K-adaptor* TAC*'s LM adaptor parameters . If $\forall \mathbf{z}_{k,1} \in \Sigma^*, \mathbf{z}_{k,2} \in \Sigma^*, \|\nabla \boldsymbol{\theta} \left(\sum \log p_{LM}(\mathbf{z}_{k,2} \mid \mathbf{z}_{k,1}; \boldsymbol{\theta})\right)\|_\infty \leq G$, then $\nabla_{\boldsymbol{\theta}} \log \mathcal{Z}_{\boldsymbol{\theta}} \leq 2G(1 - \mathcal{Z}_{\boldsymbol{\theta}})$.*

Theorems 1 and 2 provide theoretical assurance that if the model achieves high type compliance as we optimize for $\mathcal{L}'(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}) + \log \mathcal{Z}_{\boldsymbol{\theta}}$, the TACSTaR M-step update approaches true MLE update. Empirically, we observe TACSTaR rapidly drives $\mathcal{Z}_{\boldsymbol{\theta}}$ towards 1 (§4.4).

3.2 AMORTIZED TACSTAR

Amortized TACSTaR (Algorithm 4) generalize the 'fallback' rationalization heuristic in TACSTaR as parametric inference networks (Kingma & Welling, 2014; Mnih & Gregor, 2014), jointly trained to approximate the true posterior given observed input and outputs. By learning to propose better, task-adapted latent variable configurations, Amortized TACSTaR can hopefully lead to more efficient training and potentially better performance of the model TAC. For model TAC C with nodes $\mathbf{z}_1 \dots \mathbf{z}_M$, we construct an inference network TAC C' with nodes $\mathbf{z}'_1 \dots \mathbf{z}'_M$, which is trained alongside with C. In this work, we construct $\mathbf{z}'_2 \dots \mathbf{z}'_M$ to have the same types as $\mathbf{z}_2 \dots \mathbf{z}_M$, except for its input node $\mathbf{z}'_1$, which has a type to represent the input-output pair (x^*, y^*). Moreover, we construct C' so that every adaptor hyperedge e_k in C has a counterpart e'_k in C' that is additionally conditioned on $\mathbf{z}'_1$. We train C' alternately with C, with the goal of making the unnormalized distribution of C' over its nodes except for $\mathbf{z}'_1$ approximate the posterior over C's intermediate nodes, conditioning on (x^*, y^*) observations. Denoting the unnormalized distribution of C' as $\tilde{q}_{\boldsymbol{\phi}}$ parametrized by adaptors' parameters $\boldsymbol{\phi}$, we hope to learn $\boldsymbol{\phi}$ such that $\tilde{q}_{\boldsymbol{\phi}}(\mathbf{z}'_m \mid \mathbf{z}'_1 = \texttt{canon}((x^*, y^*))) \approx p_{\boldsymbol{\theta}}(\mathbf{z}_m \mid \mathbf{z}_1 = x^*_c, \mathbf{z}_2 = y^*_c)$, where $x^*_c = \texttt{canon}(x^*)$, $y^*_c = \texttt{canon}(y^*), \forall m \in [2..M]$. Approximating the posterior $p_{\boldsymbol{\theta}}(\mathbf{z}_m \mid \mathbf{z}_1 = \texttt{canon}(x^*), \mathbf{z}_2 = \texttt{canon}(y^*))$ as $\hat{p}$ using self-normalized multiple importance sampling (Veach & Guibas, 1995), we optimize $\boldsymbol{\phi}$ to minimize $\mathrm{KL}[\hat{p}||\tilde{q}_{\boldsymbol{\phi}}]$ following Bornschein & Bengio (2014); Lin & Eisner (2018).

4 EXPERIMENTS

To empirically validate TAC models, we conduct QA, code-like structured generation, and classification experiments on subsets of MGSM (Shi et al., 2023), FinQA (Chen et al., 2021), and MuSR (Sprague et al., 2024b) datasets,[10] adapting both instruction-tuned Gemma 7B and Gemma 2 27B (referred to as `gemma-1.1-7b-it` and `gemma-2-27b-it`) (Team et al., 2024), and Qwen 3 8B models (`Qwen3-8B`) (Yang et al., 2025). We aim to answer the following research questions:

- **(§4.2) Are TACs competitive against existing approaches?** TACs differ from existing LM adaptation approaches in two major ways: 1) TACs support gradient-based learning in a unified probabilistic programming framework (when compared against prior prompt optimization-focused LM programming frameworks such as DSPy); and 2)

[9] We refer the reader to §D for proofs of formal statements in this section.

[10] We defer the study of how different TAC patterns affect performance to §B, where we expand our experiments to include HotPotQA tasks (Yang et al., 2018).

TACs support structured workflows by design (when compared to the original STaR algorithm). We hypothesize that such difference translates into meaningful performance improvements.

- **(§4.3) Is exploiting TACs' probabilistic flexibility effective?** Probability models (such as TACs) benefit from the decoupling of probabilistic modeling and inference procedures, allowing conditioning on additional observations *a posteriori*. We evaluate whether exploiting this flexibility is effective in two scenarios: 1) We compare Amortized TACSTaR (§3.2), which conditions on the output variable to learn a better proposal distribution for training, against the standard (unconditioned) TACSTaR; and 2) We evaluate TACs on a classification task, comparing the performance of unconstrained generation against a renormalized classifier that evaluates and normalizes the conditional probability of each possible output.
- **(§4.4) Does the model achieve high type compliance?** A key theoretical result (§3.1) is that the soundness and near-optimality of the TACSTaR optimization strategy rely on the model learning to comply with the workflow's type constraints (*i.e.*, driving the partition function $\mathcal{Z}_{\boldsymbol{\theta}} \to 1$). As type compliance increases, the gap between the tractable unnormalized likelihood and the true normalized likelihood ($\log \mathcal{Z}_{\boldsymbol{\theta}}$) closes. We estimate how $\mathcal{Z}_{\boldsymbol{\theta}}$ over TACSTaR epochs to verify that this gap is negligible after training.

4.1 Experiment Setup

We provide an overview of our TAC and baseline DSPy setups below:

- **TACs.** We parametrize TAC adaptors to take the form of rank-1 LoRA models (Hu et al., 2022) on the attention weights, with $573,440$; $1,413,120$; and $958,464$ parameters per adaptor for `gemma-1.1-7b-it`, `gemma-2-27b-it` and `Qwen3-8B` respectively. For `parse` and `canon` implementations (§2.1), we leverage the LangFun library, which prompts LLMs to generate Python classes and objects, and parses their responses. LoRA weights are initialized ('zero-init') following Hu et al. (2022).
- **DSPy.** We conduct prompt-optimizing baseline experiments under DSPy, with base models served on vLLM. We subclass `dspy.Signature` to represent training examples, with property names and types identical to their TAC counterparts (some examples are listed in §G.2). We employ XGrammar (Dong et al., 2024) for schema-based constrained decoding for all experiments. We implement two types of reasoning workflows for all tasks: 1) the native `dspy.ChainOfThought` module, and 2) an explicitly two-step composite module that resembles **cot-cascade-structure** patterns under TACs. We experiment with various prompt optimization configurations under `dspy.MIPROv2` (Opsahl-Ong et al., 2024) and `dspy.BootstrapFewShotWithRandomSearch` (Khattab et al., 2024).

We conduct experiments of 5 reasoning-heavy tasks, on subsets from datasets MGSM[11] (Shi et al., 2023), FinQA (Yang et al., 2018), HotPotQA (Yang et al., 2018) and MuSR (Sprague et al., 2024b) respectively. Details of experiment setup are described in §G.

4.2 Comparison against prompt-optimizing and untyped STaR baselines.

Figure 2 lists MGSM, MGSM-SymPy, FinQA, and MuSR results from best-performing TACs and DSPy models. In addition, we compare the untyped (original) STaR against typed TAC results on MGSM on Gemma models.

TACs are competitive against prompt-optimizing baseline methods. We observe that TACs consistently and significantly outperform DSPy baselines in every setting. The performance gap is especially wide when 1) the base model is smaller, and 2) the task involves structured inputs (FinQA) or structured outputs (MGSM-SymPy).[12]

[11] The MGSM-SymPy task uses the same problems of MGSM, but additionally restrict the outputs to be rational expressions under SymPy. This variant was specifically included to test the framework's ability to generate and comply with highly structured, code-like output.

[12] We also compare between TACSTaR-adapted and un-adapted models on the same LangFun prompts in §B.2, and find that TACSTaR consistently outperforms the un-adapted counterparts.

Base Model	DSPy	TAC
`gemma-1.1-7b-it`	0.7%	**9.7%**
`gemma-2-27b-it`	12.7%	**34.0%**
`Qwen3-8B`	12.0%	**24.7%**

(a) FinQA

Base Model	DSPy	TAC
`gemma-1.1-7b-it`	36.5%	**62.6%**
`gemma-2-27b-it`	51.5%	**65.0%**
`Qwen3-8B`	61.5%	**63.7%**

(b) MuSR

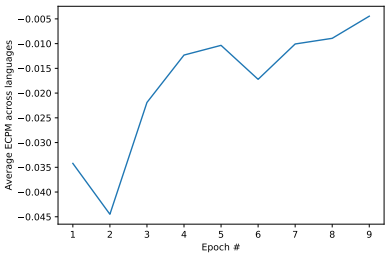
Base Model	DSPy	TAC	STaR
`gemma-1.1-7b-it`	1.6%	**27.3%**	10.5%
`gemma-2-27b-it`	81.9%	**82.2%**	76.9%

(c) MGSM

Base Model	DSPy	TAC
`gemma-2-27b-it`	57.1%	**75.9%**

(d) MGSM-SymPy

Figure 2: Comparison between best performing prompt-optimizing methods under DSPy and TACs (full results can be found in Sections H to L). We report the best DSPy result for each task.



(a) Average estimate $\log \mathcal{Z}_{\boldsymbol{\theta}}$ over validation set inputs versus # of TACSTaR epochs over MGSM languages. Note that later epochs (as early as epoch 5) do not have samples from all languages, as some languages early-stopped.

At the end of epoch	Failure rate
1	83.0%
2	1.0%
3	1.6%
4	0.4%

(b) Average MGSM training data parsing failure rate vs # of epochs of TACSTaR on `gemma-1.1-7b-it`. The pattern is **cot-cascade-structure**.

Figure 3: Type compliance during TAC training.

TACSTaR compares favorably against the original STaR algorithm on unstructured data. On the MGSM task (Fig. 2c), the original (untyped) STaR algorithm scored an average accuracy of 76.9 and 10.5 (from `gemma-2-27b-it` and `gemma-1.1-7b-it` respectively), lower than variants of reasoning TAC patterns on the same dataset. This demonstrates that the structured, typed approach of TACs improves performance over the untyped STaR baseline.

4.3 Flexible Posterior Inference Helps TAC Performance.

Amortized inference at training time is effective. The Amortized TACSTaR algorithm (§3.2) brings consistent improvement over vanilla TACSTaR on 3 tasks (Fig. 4a). Notably, the gains are most substantial on FinQA (+5.7 points). This suggests that amortized inference is particularly valuable for complex tasks where the initial sampling or fixed rationalization heuristics struggle to find valid latent traces, allowing the model to learn a more effective inference strategy.

Classification with renormalized posterior at inference time is effective. We renormalize importance sampling estimates (Eq. (4)) to estimate the output label posterior $p_{\boldsymbol{\theta}}(\mathbf{z}_2 \mid \mathbf{z}_1)$ for the MuSR classification task, and

Task	TACSTaR	Amortized TACSTaR
MGSM	82.2	82.4
FinQA	36.0	41.7
HotPotQA	32.0	34.0

(a) Comparison between TACSTaR and Amortized TACSTaR on **cot-cascade-structure** / `gemma-2-27b-it`.

Base Model	Cla.	Gen.
`gemma-1.1-7b-it`	62.6	62.1
`gemma-2-27b-it`	65.0	51.6

(b) Comparison between classification and unconstrained generation results on MuSR.

Figure 4: Comparison between 'default' and more informative inference methods.

output the label with highest probability. Figure 4b shows that the renormalized-posterior classifier outperforms unconstrained generation on both `gemma-1.1-7b-it` and `gemma-2-27b-it` base models.

4.4 TAC MODELS RAPIDLY ACHIEVE HIGH TYPE COMPLIANCE.

We argued in §3.1 that optimizing the unnormalized likelihood drives the model towards structural compliance. The average MGSM parsing error rate during training (Fig. 3b) suggests that TACs learn compliance fast. We further empirically verify this by estimating the partition function $\mathcal{Z}_{\boldsymbol{\theta}}$ — which represents the total probability mass the model assigns to type-compliant outputs (the Estimated Compliant Probability Mass, ECPM) — throughout training. We estimate $\log \mathcal{Z}_{\boldsymbol{\theta}}$ on the validation sets of the MGSM benchmark during training of the **cot-cascade-structure** pattern on `gemma-1.1-7b-it`. We sample 100 generations of entire traces without type-compliant masking per input with temperature $= 1$, top-p $= 1$, and top-k set to the vocabulary size. Figure 3a shows that the model rapidly learns to comply with the type constraints. The average $\log \mathcal{Z}_{\boldsymbol{\theta}}$ approaches -0.005 by epoch 9, corresponding to an ECPM of $\exp(-0.005) \approx 99.5\%$, and thus confirms that the degree of misspecification $(1 - Z_{\boldsymbol{\theta}})$ is negligible. Since the difference between unnormalized and normalized likelihood gradients is bounded by a multiplicative factor of $(1 - Z_{\boldsymbol{\theta}})$ (Theorem 2), our empirical estimates imply that the difference is indeed small at the end of training, and TACSTaR M-step (§3.1) approaches the true MLE update. Moreover, since $\log \mathcal{Z}_{\boldsymbol{\theta}}$ is the difference between normalized and unnormalized likelihoods, the small magnitude suggests it is practical to do model selection with unnormalized likelihood directly, after a few epochs of training.

5 RELATED WORK

The challenge of adapting LLMs to complex problems involving structured workflows and type constraints intersects with several lines of research, including programmatic LM workflows, probabilistic programming, parameter-efficient fine-tuning, and constrained decoding. We defer a more extensive survey to §A.

6 CONCLUSION

We have presented Type-Compliant Adaptation Cascades (TACs), a novel probabilistic programming framework designed to empower ML practitioners to design trainable workflows that adapt to data. Our findings demonstrate that TACs' gradient-based learning paradigm is highly effective, consistently outperforming strong prompt-optimization baselines. Moreover, we also find flexible posterior inference of TACs at both training and inference time help with performance. We also find that empirically, the model learns to comply with type constraints fast in training, justifying the assumptions in our theoretical results. These results underscore the versatility and efficacy of TACs as a scalable paradigm for adapting to complex, reasoning-heavy tasks.

REFERENCES

David Belanger and Andrew McCallum. Structured prediction energy networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pp. 983–992. JMLR.org, 2016.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi: 10.1145/3591300. URL https://doi.org/10.1145/3591300.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Guiding llms the right way: fast, non-invasive constrained generation. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. *J. Mach. Learn. Res.*, 20:28:1–28:6, 2019. URL http://jmlr.org/papers/v20/18-403.html.

Jörg Bornschein and Yoshua Bengio. Reweighted wake-sleep. *CoRR*, abs/1406.2751, 2014. URL https://api.semanticscholar.org/CorpusID:10872458.

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 (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Bowen Cao, Deng Cai, Zhisong Zhang, Yuexian Zou, and Wai Lam. On the worst prompt performance of large language models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=Mi853QaJx6.

Harrison Chase. LangChain, October 2022. URL https://github.com/langchain-ai/langchain.

Zhiyu Chen, Wenhu Chen, Charese Smiley, Sameena Shah, Iana Borova, Dylan Langdon, Reema Moussa, Matt Beane, Ting-Hao Huang, Bryan Routledge, and William Yang Wang. FinQA: A dataset of numerical reasoning over financial data. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 3697–3711, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.300. URL https://aclanthology.org/2021.emnlp-main.300.

Shay B. Cohen, Carlos Gómez-Rodríguez, and G. Satta. Elimination of spurious ambiguity in transition-based dependency parsing. *ArXiv*, abs/1206.6735, 2012. URL https://api.semanticscholar.org/CorpusID:15438603.

David Dohan, Winnie Xu, Aitor Lewkowycz, Jacob Austin, David Bieber, Raphael Gontijo Lopes, Yuhuai Wu, Henryk Michalewski, Rif A. Saurous, Jascha Sohl-dickstein, Kevin Murphy, and Charles Sutton. Language model cascades, 2022. URL https://arxiv.org/abs/2207.10342.

Yixin Dong, Charlie F Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. Xgrammar: Flexible and efficient structured generation engine for large language models. *Proceedings of Machine Learning and Systems 7*, 2024.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured NLP tasks without finetuning. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 10932–10952, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.674. URL https://aclanthology.org/2023.emnlp-main.674/.

Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. Generating structured outputs from language models: Benchmark and studies, 2025. URL https://arxiv.org/abs/2501.10868.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, chapter 18. MIT Press, 2016. http://www.deeplearningbook.org.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *ICLR*. OpenReview.net, 2020. URL http://dblp.uni-trier.de/db/conf/iclr/iclr2020.html#HoltzmanBDFC20.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 2790–2799. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/houlsby19a.html.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *ICLR*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Yasaman Jafari, Dheeraj Mekala, Rose Yu, and Taylor Berg-Kirkpatrick. MORL-prompt: An empirical analysis of multi-objective reinforcement learning for discrete prompt optimization. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 9878–9889, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.577. URL https://aclanthology.org/2024.findings-emnlp.577/.

Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive NLP. *arXiv preprint arXiv:2212.14024*, 2022.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines. 2024.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

Vijay Konda and John Tsitsiklis. Actor-critic algorithms. In S. Solla, T. Leen, and K. Müller (eds.), *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999. URL https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pp. 282–289, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1558607781.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 3045–3059, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.243. URL https://aclanthology.org/2021.emnlp-main.243/.

Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (eds.), *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 4582–4597, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.353. URL https://aclanthology.org/2021.acl-long.353/.

Chu-Cheng Lin and Jason Eisner. Neural particle smoothing for sampling from conditional sequence models. In Marilyn Walker, Heng Ji, and Amanda Stent (eds.), *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 929–941, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1085. URL https://aclanthology.org/N18-1085/.

Chu-Cheng Lin, Aaron Jaech, Xin Li, Matthew R. Gormley, and Jason Eisner. Limitations of autoregressive models and their alternatives. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (eds.), *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 5147–5173, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.405. URL https://aclanthology.org/2021.naacl-main.405.

Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. P-tuning: Prompt tuning can be comparable to fine-tuning across scales and tasks. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (eds.), *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 61–68, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-short.8. URL https://aclanthology.org/2022.acl-short.8/.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf.

Arya McCarthy, Hao Zhang, Shankar Kumar, Felix Stahlberg, and Ke Wu. Long-form speech translation through segmentation with finite-state decoding constraints on large language models. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 247–257, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.19. URL https://aclanthology.org/2023.findings-emnlp.19/.

Andriy Mnih and Karol Gregor. Neural variational inference and learning in belief networks. In Eric P. Xing and Tony Jebara (eds.), *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pp. 1791–1799, Bejing, China, 22–24 Jun 2014. PMLR. URL https://proceedings.mlr.press/v32/mnih14.html.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.525. URL https://aclanthology.org/2024.emnlp-main.525/.

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=KmtVD97J43e.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7957–7968, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.494. URL https://aclanthology.org/2023.emnlp-main.494/.

Daniel Ritchie, Paul Guerrero, R. Kenny Jones, Niloy J. Mitra, Adriana Schulz, Karl D. D. Willis, and Jiajun Wu. Neurosymbolic models for computer graphics. *Computer Graphics Forum*, 42(2):545–568, 2023. doi: https://doi.org/10.1111/cgf.14775. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14775.

Roni Rosenfeld. Two Decades of Statistical Language Modeling: Where Do We Go From Here? 6 2018. doi: 10.1184/R1/6611138.v1. URL https://kilthub.cmu.edu/articles/journal_contribution/Two_Decades_of_Statistical_Language_Modeling_Where_Do_We_Go_From_Here_/6611138.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

Freda Shi, Mirac Suzgun, Markus Freitag, Xuezhi Wang, Suraj Srivats, Soroush Vosoughi, Hyung Won Chung, Yi Tay, Sebastian Ruder, Denny Zhou, Dipanjan Das, and Jason Wei. Language models are multilingual chain-of-thought reasoners. In *ICLR*, 2023.

Dilara Soylu, Christopher Potts, and Omar Khattab. Fine-tuning and prompt optimization: Two great steps that work better together. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 10696–10710, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.597. URL https://aclanthology.org/2024.emnlp-main.597/.

Zayne Sprague, Fangcong Yin, Juan Diego Rodriguez, Dongwei Jiang, Manya Wadhwa, Prasann Singhal, Xinyu Zhao, Xi Ye, Kyle Mahowald, and Greg Durrett. To cot or not to cot? chain-of-thought helps mainly on math and symbolic reasoning, 2024a. URL https://arxiv.org/abs/2409.12183.

Zayne Rea Sprague, Xi Ye, Kaj Bostrom, Swarat Chaudhuri, and Greg Durrett. MuSR: Testing the limits of chain-of-thought with multistep soft reasoning. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=jenyYQzue1.

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.

Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. Deep probabilistic programming. In *International Conference on Learning Representations*, 2017.

Luong Trung, Xinbo Zhang, Zhanming Jie, Peng Sun, Xiaoran Jin, and Hang Li. ReFT: Reasoning with reinforced fine-tuning. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7601–7614, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.410. URL https://aclanthology.org/2024.acl-long.410/.

Eric Veach and Leonidas J. Guibas. Optimally combining sampling techniques for monte carlo rendering. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pp. 419–428, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917014. doi: 10.1145/218380.218498. URL https://doi.org/10.1145/218380.218498.

Greg C. G. Wei and Martin A. Tanner. A monte carlo implementation of the em algorithm and the poor man's data augmentation algorithms. *Journal of the American Statistical Association*, 85:699–704, 1990. URL https://api.semanticscholar.org/CorpusID:123027134.

Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V Le. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*, 2022a. URL https://openreview.net/forum?id=gEZrGCozdqR.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Huai hsin Chi, F. Xia, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *ArXiv*, abs/2201.11903, 2022b. URL https://api.semanticscholar.org/CorpusID:246411621.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, May 1992. ISSN 0885-6125. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative ai by backpropagating language model feedback. *Nature*, 639:609–616, 2025.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 15476–15488. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/639a9a172c044fbb64175b5fad42e9a5-Paper-Conference.pdf.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=92gvk82DE-.

APPENDICES

Program View	Probabilistic View
τ-typed object	Random variable $\in \Sigma^*$ restricted to strings $\in \text{valid}(\tau)$
LM adaptor with weights $\boldsymbol{\theta}$, with output restricted to τ-typed objects	Unnormalized conditional distribution $p_{LM}(\mathbf{z}_t \mid \mathbf{z}_s; \boldsymbol{\theta})\mathbb{I}(\mathbf{z}_t \in \text{valid}(\tau))$
Deterministic algorithm $f : \tau_i \to \tau_o$	Degenerate distribution $\delta_{\texttt{canon}(f(\texttt{parse}(x,\tau_i)))}(y)$
`parse` and `canon` functions that convert typed objects to/from LM inputs/outputs	Measurable maps between object domain $\mathcal{O}$ and string domain Σ^*
Executing a workflow to obtain $\mathbf{z}_{1\ldots M}$	Sampling from joint unnormalized probability $\tilde{p}_{\boldsymbol{\theta}}(\mathbf{z}_{1\ldots M}) = \prod_k \tilde{p}_{\boldsymbol{\theta}}(\mathbf{z}_{T_k} \mid \mathbf{z}_{S_k})$
Probability that a stochastic workflow succeeds	$\mathcal{Z}_{\boldsymbol{\theta}} = \Pr_{p_{\boldsymbol{\theta}}}(\text{all nodes are valid})$

Table 1: Dual semantics: how TAC concepts map between their program and probabilistic views.

A BACKGROUND AND RELATED WORK

Programmatic LM workflows. A large body of work exposes LMs through *programmed* pipelines as typed or templated modules, with declarative constraints and optimizers, such as DSPy (Khattab et al., 2022; 2024), LMQL (Beurer-Kellner et al., 2023), and LangChain (Chase, 2022). These systems typically *specify* structure and then tune prompts or few-shot exemplars. They do not cast the entire workfrlow as a single probabilistic object with learnable continuous parameters, and a likelihood objective. While there have been proposals that optimized weights under such programmatic pipelines (such as `BetterTogether` (Soylu et al., 2024)), TACs differs fundamentally in its principled yet optimization-friendly probabilistic formulation, which enables both theoretically justified training methods (§3.1) and advanced inference techniques (§3.2).

Probabilistic programming and structured prediction. Probabilistic programming languages tailored for machine learning, such as Edward (Tran et al., 2017) and Pyro (Bingham et al., 2019), combine differentiable components with stochastic control flow. On the other hand, classical structured prediction (Lafferty et al., 2001; Belanger & McCallum, 2016) provides tools for handling global constraints in unnormalized models. Our formulation connects these threads to LM workflows: each typed hyperedge is an *unnormalized* conditional whose **type compliance** functions as a partition function term $\mathcal{Z}_{\boldsymbol{\theta}} \leq 1$. This distinct perspective allows us to train with a tractable objective, whose bias vanishes as type compliance rises.

Problem-solving strategies and adapting for reasoning. Techniques like Chain-of-Thought (CoT) (Wei et al., 2022b) and Self-Refine (Madaan et al., 2023) leverage prompting to elicit intermediate problem-solving steps or iterative improvements from LMs, often boosting performance on complex tasks. Methods such as STaR (Zelikman et al., 2022) and ReFT (Trung et al., 2024) further adapted the LM to reason. We adopt the spirit of STaR, but place it inside a hypergraph to propose typed and multi-step rationalizations (§3.1). We also introduce an amortized variant that learns to propose rationalizations, rather than relying solely on heuristics (§3.2).

Constrained and schema-aware decoding. To improve output reliability, various methods enforce grammar-based constraints during LLM generation (Poesia et al., 2022; Geng et al., 2023; McCarthy et al., 2023; Beurer-Kellner et al., 2024; Geng et al., 2025) have been proposed. These methods generally modify *local* conditional distributions over next tokens, to mask out continuations that are incompatible with the given input and grammar. In contrast, our objective learns parameters so that type-compliant trajectories carry increasing probability mass globally, improving validity and task accuracy.

Parameter-efficient adaptation. LoRA and related PEFT methods (Houlsby et al., 2019; Hu et al., 2022; Li & Liang, 2021; Lester et al., 2021; Liu et al., 2022) enable light-weight adaptation. We use small adaptors to highlight data-efficiency and show that gains stem from *typed workflow learning* rather than sheer capacity.

Connection to Reinforcement Learning. The TACSTaR training procedure (§3.1) can be viewed through the lens of policy optimization. As Zelikman et al. (2022) observed, the STaR objective closely resembles the REINFORCE algorithm (Williams, 1992). Similarly, the M-step in the TACSTaR algorithm can be interpreted as optimizing the TAC workflow policy under REINFORCE, where a binary reward is assigned upon successfully generating the correct output.

We adopt the MC-EM framing as it provides a principled approach for likelihood maximization in the presence of annotated output data. While more advanced RL techniques (*e.g.*, PPO (Schulman et al., 2017) or actor-critic methods (Konda & Tsitsiklis, 1999)) work with non-binary reward functions, they often introduce complexity, such as training value functions, which are difficult to estimate over complex, typed latent spaces. Furthermore, the exploration challenge often faced by policy gradient methods in sparse reward settings is significantly mitigated by both the rationalization heuristic and the inference network in Amortized TACSTaR (§3.2) in the E-step. This mechanism effectively guides the sampling process towards successful trajectories using the known outputs — a technique specific to this supervised adaptation context.

B Additional Studies on Workflow Pattern Design

In this section, we conduct additional experiments that vary the pattern structures, and evaluate how such changes affect performance. Specifically, we would like to answer the following questions:

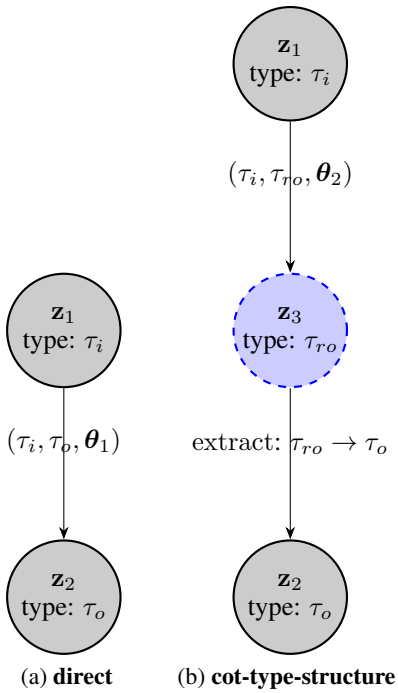
- **(§B.2) Is adaptation with reasoning workflows effective?** The TAC framework gives practitioners great freedom in designing a workflow that reason in the process. We hypothesize that adapting with such explicit structures improves performance on tasks that require complex reasoning.
- **(§B.3) How do TAC design variations affect performance?** We evaluate how such TAC design variations for the same task affect performance.

B.1 End-to-end trainable workflows as TACs.

The declarative and flexible nature of TACs enable practitioners to rapidly implement end-to-end trainable workflows. We implement some common patterns as TACs:

- **Direct adaptation** of an LM to the downstream task without any latent structure corresponds to common supervised PEFT methods surveyed in §A. The **direct** pattern (Fig. 5a) is a singleton TAC with no latent nodes.
- **Adapting with latent rationales** corresponds to patterns that learn to generate rationales for the task at hand Zelikman et al. (2022). There are several possible TAC structure designs that incorporate rationales: for example, **cot-type-structure** (Fig. 5b) maps the input to a rationale-output typed object, from which the task output is deterministically extracted. Alternatively, **cot-cascade-structure** (Fig. 1a) introduce rationales as distinct nodes in the TAC hypergraph, which transforms into the task output under an adaptor.
- **Trainable self-refinement** refers to an end-to-end trainable variant of self-refine (Madaan et al., 2023), where the model first sketches a task output, and iteratively refine it. Without TAC, a practitioner would have to resort to manually writing tedious postprocessing functions for the intermediate results. On the other hand, the TAC counterpart **refine-structure** (Fig. 6 in §F) is straightforward.

For the MGSM-SymPy task, we experiment with the **expression-cascade-structure** pattern (Fig. 1b), which additionally imposes the constraint that the output must be a rational number represented by an arithmetic



(a) **direct** (b) **cot-type-structure**

Figure 5: Workflow patterns experimented in this paper, with increasing structural complexity from left to right. In the most complicated pattern **expression-cascade-structure** we illustrate the workflow with example node values. Dashed-boundary nodes indicate variables that are not observed at training time. And solid-boundary nodes indicate nodes with training time observable values. A main message of this work is that instead of defining workflows imperatively as fixed-parameter systems, **we treat an entire typed workflow as a single probabilistic program, whose parameters are lightweight PEFT modules, allowing end-to-end training with latent variables.**

expression tree. Such type constraints often reflect business logic (for example, we expect the MGSM dataset to have rational number answers), and may be necessary when the TAC forms a component in a larger system.

B.2 Effectiveness of Adaptation with Reasoning Workflows

To evaluate whether adaptation with reasoning workflows is effective, we compare **cot-cascade-structure**, and **refine-structure** TACs against **direct** on the 3 tasks MGSM, FinQA and HotPotQA, on base models `gemma-2-27b-it` and `gemma-1.1-7b-it`. Table 2 shows that both **cot-cascade-structure** significantly outperforms **direct** on MGSM and FinQA on both `gemma-2-27b-it` and `gemma-1.1-7b-it`. But **cot-cascade-structure** slightly underperforms **direct** on HotPotQA. These results largely agree with the meta study done by Sprague et al. (2024a), which also reported that tasks that require arithmetic and symbolic reasoning, such as MGSM and FinQA, benefit the most from CoT, while a huge portion of previous work saw that CoT degrades performance for multihop QA. However, we note that the **refine-structure** TAC (Fig. 6) consistently outperform the **direct** baseline in all 3 tasks on `gemma-2-27b-it`, showcasing the effectiveness of the adaptive refinement paradigm.

	gemma-2-27b-it			gemma-1.1-7b-it	
Dataset	direct	cot-cascade-structure	refine-structure	direct	cot-cascade-structure
MGSM	24.7	**82.2**	78.6	5.1	**27.3**
FinQA	17.3	**36.0**	23.7	3.0	**9.7**
HotPotQA	34.0	32.0	**39.0**	—	—

Table 2: Comparison between **direct** and reasoning workflows. For the MGSM dataset, we report per-language accuracies in Table 5. The difference between best performing runs and **direct** are statistically significant/marginally significant: for MGSM and FinQA $p < 0.05$ (both `gemma-2-27b-it` and `gemma-1.1-7b-it`), and for HotPotQA $p = 0.07$ under paired permutation tests. Per-language accuracy numbers of the MGSM dataset are in §H.

Task adaptation with TACSTaR is effective. To evaluate whether the efficacy of TACs can be attributed to our proposed TACSTaR method, we also compare adapted TAC workflows against those with the same hypergraph structure, but with un-adapted weights (*i.e.*, all adaptors in the TAC use base model weights). Both TACSTaR trained and un-adapted models use the same structured LangFun prompts that are similar to examples listed in §N. The significant gap between adapted and un-adapted results in Table 3 indicate that the TACSTaR algorithm is effective. Notably, un-adapted models still outperform **direct** workflows (listed in Table 2), indicating that LangFun's type-inducing prompts can invoke somewhat effective test-time computation over the TAC hypergraph structure.

Task	Structure	TACSTaR	Un-adapted
MGSM	cot-cascade-structure	82.2	45.4
MGSM	cot-type-structure	80.4	74.7
MGSM-SymPy	expression-cascade-structure	75.9	69.5
FinQA	cot-cascade-structure	36.0	13.0
HotPotQA	refine-structure	39.0	24.0

Table 3: Comparison between TACSTaR-adapted and un-adapted `gemma-2-27b-it`. The differences are all statistically significant ($p < 0.05$) under paired permutation tests.

B.3 Effects of Different TAC Designs

Decoupling rationale and output modeling helps performance. **cot-cascade-structure** (Fig. 1a) achieves a higher score than **cot-type-structure** (Fig. 5b) on the MGSM task (Table 4), suggesting that modeling the rationale and task output generation with distinct adaptors helps performance. By using distinct adaptors, the workflow allows specialization: the first adaptor focuses on reasoning, while the second specializes in synthesis, reducing the complexity burden on a single monolithic step. The positive result again highlights how the TAC formalism can help practitioners iterate and experiment with different multi-adaptor cascade designs, which would be tedious otherwise.

Robustness to Semantic Constraints. Comparing performance on MGSM and the more constrained MGSM-SymPy task reveals a key advantage of the TAC framework's robustness. As shown in Table 4, the best-performing TAC model sees a modest performance drop, from 82.2% on MGSM to 75.9% on MGSM-SymPy, when required to generate a valid symbolic expression.[13] This contrasts sharply with the prompt-optimizing baseline (Fig. 2). The best DSPy configuration experiences a much more significant degradation, plummeting from 81.9% on MGSM to

[13] Sample expressions generated under **expression-cascade-structure** are listed in §M.

just 57.1% on MGSM-SymPy. The substantially smaller performance drop for TACs underscores the brittleness of discrete prompt optimization when faced with strict structural requirements. The TAC framework's gradient-based adaptation within a typed system proves to be significantly more resilient, making it a more reliable paradigm for tasks demanding structural compliance.

	MGSM	MGSM-SymPy
cot-type-structure	**cot-cascade-structure**	**expression-cascade-structure**
80.4	82.2	75.9

Table 4: Effects of different TAC designs on the MGSM dataset, demonstrating the impact of workflow structure on performance. The **cot-cascade-structure** (which decouples rationale generation from the final answer synthesis) outperforms the monolithic **cot-type-structure**. The **expression-cascade-structure** result shows strong performance on the more constrained MGSM-SymPy task.

C ALGORITHMS

C.1 FORWARD AND BACKWARD

Algorithm 1 (`forward`) executes the probabilistic program represented by a TAC $C = (\mathbf{Z}, \mathbf{E})$. Starting from a given input node value $\mathbf{z}_1^*$, the algorithm traverses the hypergraph following a topological order, and terminates when all edges $\in \mathbf{Z}$ have been visited. `forward` takes C and $\mathbf{z}_1^*$ as input arguments. `forward` also takes the following as arguments:

- sampler configuration κ for different sampling techniques, *e.g.*, varying temperature, nucleus, and top-k sampling
- maximum number of sampling attempts

Algorithm 2 (`backward`) takes as input $(C, \mathbf{Z}^*)$, where $C = (\mathbf{Z}, \mathbf{E})$ where $\mathbf{E} = (e_1 \dots e_K)$ is a TAC, and $\mathbf{Z}^*$ are value assignments of $\mathbf{Z}$. We assume the log probability $p_{LM}(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta}_k)$ is auto-differentiable with regard to all adaptor hyperedges in a TAC. Algorithm 2 returns unnormalized log joint probabilities of $\mathbf{Z}^*$ under C: $\log \tilde{p}_{\boldsymbol{\theta}}(\mathbf{Z}^*)$, the per-node generation log probabilities $(\log p_{\boldsymbol{\theta}}(z_2 \mid \cdot) \dots \log p_{\boldsymbol{\theta}}(z_M \mid \cdot))$, and also gradients of LM adaptors: $\nabla_{\boldsymbol{\theta}_k} \log \tilde{p}_{\boldsymbol{\theta}}(\mathbf{Z}^*)$ for adaptor hyperedges' indices k. We note that `backward` is easily parallelizable: all adaptor edges can be processed at the same time.

C.2 TACSTAR

The TACSTaR algorithm (Algorithm 3) takes as input $(C, \{x_i^*, y_i^* \mid i \in [1..\mathcal{D}_{\text{train}}]\})$, where C is the TAC to train, and $\{(x_i^*, y_i^*) \mid i \in [1..\mathcal{D}_{\text{train}}]\}$ is the training dataset. As we described in §3.1, TACSTaR uses a 'fallback TAC' heuristics in hope to obtain a sample when the forward algorithm fails.

Building Fallback TAC. Given a TAC $C = (\mathbf{Z}, \mathbf{E})$ with input node and output node typed τ_i and τ_o respectively, we build its fallback TAC $C_{\text{fallback}} = (\mathbf{Z}', \mathbf{E}')$ (denoted as the function `build_fallback` in Algorithm 3) as follows:

- The input node of C_{fallback}: $\mathbf{z}_1'$ is of the product type $\tau_{io} = \tau_i \times \tau_o$, representing a data container that holds one object of type τ_i and another object of type τ_o.
- All other nodes $\in \mathbf{Z}$ have their counterpart nodes in Z' (with the same types and indices).

Algorithm 1 TAC Forward Algorithm (`forward`)

Input: TAC cascade $C = (\mathbf{Z}, \mathbf{E})$ where $\mathbf{Z} = \{\mathbf{z}_1 \dots \mathbf{z}_M\}$ and $\mathbf{E} = \{\mathbf{e}_1 \dots \mathbf{e}_K\}$, input object: $\mathbf{z}_1^*$, sampler configuration κ, N_{max} for maximum number of sampling attempts.
Output: Sampled values $(\mathbf{z}_2^*, \dots, \mathbf{z}_M^*)$.

Determine a topological ordering of edges in $\mathbf{E}$. Let the sorted hyperedges be $e'_1 \dots e'_K$.
$\mathbf{Z}^*_{\text{already\_sampled}} \leftarrow \{\mathbf{z}_1^*\}$.
for $k \in [1..K]$ **do**
    Assert the source nodes of e'_k is a subset of $\mathbf{z}_{\text{already\_sampled}}$.
    if $e'_k = (\tau_i, \tau_o, \boldsymbol{\theta})$ is a type-constrained LM adaptor **then**
        # type-constrained LM adaptors have a single source node and a single target node.
        $\boldsymbol{x} \leftarrow$ canonicalized representation of e'_k's source node.
        while number of attempts $\leq N_{\text{max}}$ **do**
            Try draw $\boldsymbol{y} \sim p_{LM}(\cdot \mid \boldsymbol{x}; \boldsymbol{\theta}, \kappa)$
            if $\texttt{parse}(\boldsymbol{y}, \tau_o) \neq$ error **then**
                $t \leftarrow$ index of e'_k's target node.
                $\mathbf{z}_t^* \leftarrow \boldsymbol{y}$
                $\mathbf{Z}^*_{\text{already\_sampled}} \leftarrow \mathbf{Z}_{\text{already\_sampled}} \cup \{\mathbf{z}_t^*\}$
                break
            end if
        end while
    else if e'_k is a deterministic algorithm f **then**
        # In this work we assume f's inputs and outputs are sorted by node index in C.
        $\mathbf{O}_{f\text{input}} \leftarrow$ parsed objects of e'_k's source nodes, sorted by node index.
        $\mathbf{O}_{f\text{output}} \leftarrow f(\mathbf{O}_{f\text{input}})$
        $\mathbf{Z}^*_{f\text{output}} \leftarrow$ canonicalized representations of objects $\in \mathbf{O}_{f\text{output}}$, sorted by node index.
        $\mathbf{Z}_{\text{already\_sampled}} \leftarrow \mathbf{Z}^*_{\text{already\_sampled}} \cup \mathbf{Z}^*_{f\text{output}}$
    end if
end for
return $\mathbf{Z}_{\text{already\_sampled}} - \{\mathbf{z}_1^*\}$.

- We copy each hyperedge $e \in \mathbf{E}$ over to $\mathbf{E}'$, connecting nodes with the same indices. In the case that e is a deterministic algorithm hyperedge, and has $\mathbf{z}_1$ as one of its source nodes, we modify the counterpart hyperedge e' to have a deterministic algorithm that first extracts the original object $\texttt{parse}(\mathbf{z}_1)$ from $\texttt{parse}(\mathbf{z}'_1)$, and then pass $\texttt{parse}(\mathbf{z}_1)$ to the original algorithm as input.

Adaptors in C_{fallback} use no-op weights, falling back to the behavior of the base model. We denote such no-op weights as $\boldsymbol{\theta}_0$. For example, Fig. 7 is the C_{fallback} for Fig. 5b.

C.3 AMORTIZED TACSTAR

The Amortized TACSTaR algorithm (Algorithm 4) builds upon Algorithm 3 to introduce an inference network TAC. While C_{fallback} used fixed no-op weights that behave identical to the base language model, Amortized TACSTaR leverages an inference network TAC C' with trainable parameters.

Building the inference network C'. Given a TAC $C = (\mathbf{Z}, \mathbf{E})$ with input node and output node typed τ_i and τ_o respectively, we build the adaptive fallback TAC$C' = (\mathbf{Z}', \mathbf{E}')$ (denoted as the function `build_infer_net` in Algorithm 4). At a high level, every adaptor hyperedge that generates latent variables in C is mapped into a

Algorithm 2 TAC Backward Algorithm (`backward`)

Input: $C = (\mathbf{Z}, \mathbf{E})$ and sample $\mathbf{Z}^* = \{\mathbf{z}_1^*, \mathbf{z}_2^*, \ldots, \mathbf{z}_M^*\}$
Output: $(\log \tilde{p}_{\boldsymbol{\theta}}(\mathbf{Z}^*), \quad (\log p_{\boldsymbol{\theta}}(z_2 \quad | \quad \cdot) \ldots \log p_{\boldsymbol{\theta}}(z_M \quad | \quad \cdot)), \quad \{\nabla_{\boldsymbol{\theta}_k} \log \tilde{p}_{\boldsymbol{\theta}}(\mathbf{Z}^*) \quad | \quad e_k \quad \in$ $\mathbf{E}$ is an adaptor hyperedge$\})$

Initialize log-probability accumulator $\mathcal{L} \leftarrow 0$.
for each LM adaptor hyperedge $e_k = (\tau_i, \tau_o, \boldsymbol{\theta}_k)$ **do**
    Let $\mathbf{z}_i^* \in \mathbf{Z}^*$, $\mathbf{z}_o^* \in \mathbf{Z}^*$ be the sample value of e_k's input and output nodes $\mathbf{z}_i$ (typed τ_i) and $\mathbf{z}_o$ respectively.
    $(\ell, \mathbf{g}_k) \leftarrow$ `peft_backward`$(\log p_{LM}(\mathbf{z}_o^* \mid$ `canon`$($`parse`$(\mathbf{z}_i^*, \tau_i)); \boldsymbol{\theta})$.
    $\mathcal{L} \leftarrow \mathcal{L} + \ell$
    keep track of ℓ by its node index.
end for
 # For nodes from deterministic hyperedges, set log prob to 0 **as they have no learnable parameters.**
return $(\mathcal{L}, (\log p_{\boldsymbol{\theta}}(z_2 \mid \cdot) \ldots \log p_{\boldsymbol{\theta}}(z_M \mid \cdot)), \{\mathbf{g}_k \mid e_k \in \mathbf{E}$ is an adaptor hyperedge$\})$.

counterpart in C' that also depends on both observed a τ_i-typed input and a τ_o-typed output, now encoded as $\mathbf{z}'_1$, typed τ_{io}. Specifically we build C' with the following procedure:

- The input node of C': $\mathbf{z}'_1$ is of the product type $\tau_{io} = \tau_i \times \tau_o$, as with `build_fallback`.
- All nodes $\in \mathbf{Z}$ have their counterpart nodes in Z' (with the same types and indices), except for $\{\mathbf{z}_1, \mathbf{z}_2\}$.[14]
- For each hyperedge $e \in \mathbf{E}$,
  - In the case that e is a deterministic algorithm hyperedge, and has $\mathbf{z}_1$ as one of its source nodes, we add a counterpart hyperedge e' that connect counterpart nodes in $\mathbf{Z}'$, with its deterministic algorithm modified to typecheck, as `build_fallback`.
  - Otherwise, e is an adaptor hyperedge. Denoting its source node as $\mathbf{z}_s$ and target node as $\mathbf{z}_t$:
    * If $\mathbf{z}_t = \mathbf{z}_2$, we continue since $\mathbf{z}_t$ has no counterpart C'.
    * If $\mathbf{z}_s = \mathbf{z}_1$ and $\mathbf{z}_t \neq \mathbf{z}_2$, we add a counterpart hyperedge $e' = (\tau_s, \tau_t, \boldsymbol{\theta}_{\text{new}})$ connecting counterpart nodes $\mathbf{z}'_s$ and $\mathbf{z}'_t$. $\boldsymbol{\theta}_{\text{new}}$ indicates the parameter vector of a new LM adaptor.
    * Otherwise, $\mathbf{z}_s \neq \mathbf{z}_1$ and $\mathbf{z}_t \neq \mathbf{z}_2$. In this case, we create e' to be an adaptor that is conditioned on both $\mathbf{z}'_s$ and $\mathbf{z}'_1$. To achieve this goal, we introduce into C' a helper node $\mathbf{z}''_s$ typed $\tau_{ios} = \tau_i \times \tau_o \times \tau_s$, and a helper hyperedge e'' that has source nodes $\{\mathbf{z}'_1, \mathbf{z}'_s\}$, and target node $\{\mathbf{z}''_s\}$. e'' is a deterministic edge that combines values in $\mathbf{z}'_1$ and $\mathbf{z}'_s$ into the 3-object container $\mathbf{z}''_s$. Finally, we add e' that connects $\mathbf{z}''_s$ to $\mathbf{t}$ as the adaptor transformation $(\tau_{ios}, \tau_t, \boldsymbol{\theta}_{\text{new}})$, where $\boldsymbol{\theta}_{\text{new}}$ again indicates the parameter vector of a new LM adaptor.

Adaptors in C' are new adaptors. And we train C alternately with C' in Algorithm 4. The algorithm to train C' is listed in Algorithm 5.

C.4 Updating C'

We train the inference network C' to better approximate the posterior distribution defined by C alternately (§3.2). In other words, we update adaptor parameters in C' so that sampled latent variables of C' $((\hat{\mathbf{z}}_3, \ldots, \hat{\mathbf{z}}_M)$ obtained using `forward`$(C', $`canon`$(x^*), \kappa))$ follow the normalized distributions under C (obtained using `backward`$(C, ($`canon`$(x^*), $`canon`$(y^*), \hat{\mathbf{z}}_3, \ldots, \hat{\mathbf{z}}_M))))$. To promote diversity of samples, we additionally obtain samples from C_{fallback} (§C.2). Let $\mathbf{Z} = (\mathbf{z}_3^*, \ldots, \mathbf{z}_M^*)$ be a sample out of G collected samples

[14] We arbitrarily designate a node $\in \mathbf{Z}'$ that does not have an outgoing hyperedge as the output node for syntactic conformity.

Algorithm 3 TACSTaR Training Algorithm

Input: Training pairs $\mathcal{D}_{\text{train}} = \{(x_i^*, y_i^*) \mid i \in [1..|\mathcal{D}_{\text{train}}|]\}$, TAC C, sampler configuration κ.

$C_{\text{fallback}} \leftarrow$ build_fallback(C)
for epoch in [1..num_epochs] **do**
  $S \leftarrow \{\}$ **# Successful samples**
  for training pair $(x^*, y^*) \in \mathcal{D}_{\text{train}}$ **do**
    $\mathbf{z}_1^* \leftarrow \texttt{canon}(x^*)$
    # E-step (Sampling Latent Variables):
    $(\hat{\mathbf{z}}_2 \dots \hat{\mathbf{z}}_M) \leftarrow \text{Forward}(C, \mathbf{z}_1^*)$.
    # Filtering (Validity Check):
    Initialize error_flag $\leftarrow$ false.
    Set error_flag $\leftarrow$ true if errors in E-step or $\texttt{parse}(\hat{\mathbf{z}}_2) \neq y^*$.
    # Heuristics Fallback (Addressing Forward Failure):
    if error_flag is true **then**
      $\mathbf{z}_1'^* \leftarrow \texttt{canon}((x^*, y^*))$
      $(\hat{\mathbf{z}}_2' \dots \hat{\mathbf{z}}_M') \leftarrow \texttt{forward}(C_{\text{fallback}}, \mathbf{z}_1'^*)[0]$.
      if no error was raised and $\texttt{parse}(\hat{\mathbf{z}}_2') = y^*$ **then**
        $(\hat{\mathbf{z}}_2 \dots \hat{\mathbf{z}}_M) \leftarrow (\hat{\mathbf{z}}_2' \dots \hat{\mathbf{z}}_M')$
        Set error_flag $\leftarrow$ false.
      end if
    end if
    if error_flag is false **then**
      $S \leftarrow S \cup \{(\mathbf{z}_1^*, \hat{\mathbf{z}}_2 \dots \hat{\mathbf{z}}_M)\}$
    end if
  end for
  # M-step (Parameter Update):
  for $(\mathbf{z}_1^*, \hat{\mathbf{z}}_2 \dots \hat{\mathbf{z}}_M) \in S$ **do**
    $\mathbf{G} \leftarrow \texttt{backward}(C, (\mathbf{z}_1^*, \hat{\mathbf{z}}_2 \dots \hat{\mathbf{z}}_M))[2]$
    $\texttt{optimize}(C, \mathbf{G})$
  end for
end for

$(\mathbf{Z}^{(1)}, \dots, \mathbf{Z}^{(G)})$ from C_{fallback} and C'. We approximate the posterior probability of $\mathbf{Z}$ under C, conditioning on $\mathbf{z}_1^* = \texttt{canon}(x^*)$, $\mathbf{z}_2^* = \texttt{canon}(y^*)$ under the balance heuristic (Veach & Guibas, 1995) as

$$\hat{p}_{\text{posterior}}(\mathbf{Z}) \propto \frac{(N_{\text{fallback}} + N_{\text{infer}})\tilde{p}_{\text{model}}}{N_{\text{fallback}} p_{\text{fallback}} + N_{\text{infer}} p_{\text{infer}}}, \tag{6}$$

where $\tilde{p}_{\text{model}} = \tilde{p}_C(\mathbf{z}_1^*, \mathbf{z}_2^*, \mathbf{z}_3^*, \dots, \mathbf{z}_M^*)$, $p_{\text{fallback}} = \prod_{m=3}^{M} p_{LM}(\mathbf{z}_m^* \mid \mathbf{z}_m^*\text{'s source node}; \boldsymbol{\theta}_0)$, and $p_{\text{infer}} = \prod_{m=3}^{M} p_{LM}(\mathbf{z}_m^* \mid \mathbf{z}_m^*\text{'s source node}; \boldsymbol{\theta}_{\text{new}})$. These values are all obtained using the backward algorithm.[15] We denote the number of samples attempted (including errors) on $C_{\text{fallback}} = N_{\text{fallback}}$, the number of samples attempted (including errors) on $C' = N_{\text{infer}}$. $\hat{p}_{\text{posterior}}$ is normalized over the mixture so that $\sum_{g=1}^{G} \hat{p}_{\text{posterior}}(\mathbf{Z}^{(g)}) = 1$.

Algorithm 5 updates adaptors in C' to bring its unnormalized distribution closer to Eq. (6). Since the self-normalized approximation of the posterior distribution is consistent but biased, we require minimum numbers of samples from C' and C_{fallback}.

[15] backward algorithm as presented in this work computes both gradients and probabilities. In our implementation we do not compute gradients when they are not needed; but we omit this subtlety in Algorithm 2.

Algorithm 4 Amortized TACSTaR Training Algorithm

Input: Training pairs $\mathcal{D}_{\text{train}} = \{(x_i^*, y_i^*) \mid i \in [1..|\mathcal{D}_{\text{train}}|]\}$, TAC C, sampler configuration κ.

$C' \leftarrow$ build_infer_net(C)
for epoch in $[1..\texttt{num\_epochs}]$ **do**
  $S \leftarrow \{\}$ **# Successful samples**
  for training pair $(x^*, y^*) \in \mathcal{D}_{\text{train}}$ **do**
    $\mathbf{z}_1^* \leftarrow \texttt{canon}(x^*)$
    # E-step (Sampling Latent Variables):
    $(\hat{\mathbf{z}}_2 \dots \hat{\mathbf{z}}_M) \leftarrow$ Forward$(C, \mathbf{z}_1^*)$.
    # Filtering (Validity Check):
    Initialize `error_flag` $\leftarrow$ false.
    Set `error_flag` $\leftarrow$ true if errors in E-step or $\texttt{parse}(\hat{\mathbf{z}}_2) \neq y^*$.
    # Heuristics Fallback (Addressing Forward Failure):
    if `error_flag` is true **then**
      $\mathbf{z}_1'^* \leftarrow \texttt{canon}((x^*, y^*))$
      $(\hat{\mathbf{z}}_2' \dots \hat{\mathbf{z}}_M') \leftarrow \texttt{forward}(C_{\text{fallback}}, \mathbf{z}_1'^*)[0]$.
      if no error was raised and $\texttt{parse}(\hat{\mathbf{z}}_2') = y^*$ **then**
        $(\hat{\mathbf{z}}_2 \dots \hat{\mathbf{z}}_M) \leftarrow (\hat{\mathbf{z}}_2' \dots \hat{\mathbf{z}}_M')$
        Set `error_flag` $\leftarrow$ false.
      end if
    end if
    if `error_flag` is true **then**
      $(\hat{\mathbf{z}}_3 \dots \hat{\mathbf{z}}_M) \leftarrow \texttt{forward}(C', \mathbf{z}_1^*)[0]$
      Set `error_flag` $\leftarrow$ false if no errors in previous step.
    end if
    if `error_flag` is false **then**
      $S \leftarrow S \cup \{(\mathbf{z}_1^*, \mathbf{z}_2^*, \hat{\mathbf{z}}_3, \dots \hat{\mathbf{z}}_M)\}$
    end if
  end for
  # M-step (Parameter Update):
  for $(\mathbf{z}_1^*, \hat{\mathbf{z}}_2 \dots \hat{\mathbf{z}}_M) \in S$ **do**
    $\mathbf{G} \leftarrow \texttt{backward}(C, (\mathbf{z}_1^*, \hat{\mathbf{z}}_2 \dots \hat{\mathbf{z}}_M))[2]$
    $\texttt{optimize}(C, \mathbf{G})$
  end for
  $C' \leftarrow$ update inference network C' (§C.4).
end for

D Formal Statements and Proofs Regarding Type Compliance

Well-specifiedness. Let $C = (\mathbf{Z}, \mathbf{E})$. We define well-specifiedness for TAC: we say $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1 \dots \boldsymbol{\theta}_K\}$ is well-specified if for every LM adaptor $e_k = (\tau_i, \tau_o, \boldsymbol{\theta}_k) \in \mathbf{E}$ and for every valid canonicalized string $\boldsymbol{x}$ of type τ_i, the LM distribution p_{LM} only has support over valid outputs of type τ_o. Formally, $\forall$ valid x, $\sum_{\boldsymbol{y} \in \mathcal{D}_{\text{valid}}(\tau_o)} p_{LM}(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta}_k) = 1$ iff $\boldsymbol{\theta}$ is well-specified.

We first prove that hyperedges are locally normalized (*i.e.*, the partition function is 1) when $\boldsymbol{\theta}$ is well-specified:

Lemma 1. *If $\boldsymbol{\theta}$ is well-specified, then for any hyperedge $e_k \in \mathbf{E}$ and any valid assignment $\boldsymbol{x}$ to its source nodes, the local partition function $Z_k = 1$.*

Algorithm 5 `update_infer_net`

Input: Training pair (x^*, y^*), model TAC C, sampler configuration κ, inference network C', non-adaptive fallback C_{fallback}, number of samples from C_{fallback}: G_{fallback}, number of samples from C': G_{infer} .

$\mathbf{z'}^*_1 \leftarrow \texttt{canon}((x^*, y^*))$, $\mathbf{z}^*_1 \leftarrow \texttt{canon}(x^*)$, $\mathbf{z}^*_2 \leftarrow \texttt{canon}(y^*)$.
$\mathbf{Z}_{\text{collected}} \leftarrow []$
 # In our implementation we give up and raise an error after 30 **unsuccessful attempts.**
while number of successful samples from $C_{\text{fallback}} < G_{\text{fallback}}$ **do**
    Try $(\hat{\mathbf{z}}_2, \hat{\mathbf{z}}_3, \dots \hat{\mathbf{z}}_M) \leftarrow \texttt{forward}(C_{\text{fallback}}, \mathbf{z'}^*_1, \kappa, 1)$
    if previous step succeeded **then**
        # We discard $\hat{\mathbf{z}}_2$ from C_{fallback}.
        Append $(\hat{\mathbf{z}}_3, \dots, \hat{\mathbf{z}}_M)$ to $\mathbf{Z}_{\text{collected}}$.
    end if
end while
$N_{\text{fallback}} \leftarrow$ numbers of attempts on C_{fallback}
while number of successful samples from $C' < G_{\text{infer}}$ **do**
    Try $(\hat{\mathbf{z}}_3, \dots \hat{\mathbf{z}}_M) \leftarrow \texttt{forward}(C', \mathbf{z'}^*_1, \kappa, 1)$
    if previous step succeeded **then**
        Append $(\hat{\mathbf{z}}_3, \dots, \hat{\mathbf{z}}_M)$ to $\mathbf{Z}_{\text{collected}}$.
    end if
end while
$N_{\text{infer}} \leftarrow$ numbers of attempts on C'
$G \leftarrow G_{\text{fallback}} + G_{\text{infer}}$
Assert $G = |\mathbf{Z}_{\text{collected}}|$
Compute $[\hat{p}_{\text{posterior}}(\mathbf{Z}^{(1)} \dots \hat{p}_{\text{posterior}}(\mathbf{Z}^{(G)})]$ using Eq. (6).
Sample $g \in [1..G]$ with probability proportional to $\hat{p}_{\text{posterior}}(\mathbf{Z}^{(g)}$.
$\mathbf{G} \leftarrow \texttt{backward}(C', \mathbf{Z}^{(g)})[2]$.
$\texttt{optimize}(C', \mathbf{G})$

Proof. e_k is either an LM adaptor or a deterministic algorithm:

- If e_k is an LM adaptor, $Z_k = \sum_{\boldsymbol{y}} \tilde{p}_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}; e_k) = \sum_{\boldsymbol{y} \in \text{valid}(\tau_o)} p_{LM}(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta}_k) = 1$.
- If e_k is a deterministic algorithm, by Eq. (2) $Z_k = \sum_{\boldsymbol{y}} \tilde{p}(\boldsymbol{y} \mid \boldsymbol{x}; e_k) = \tilde{p}(\texttt{canon}(f(\texttt{parse}(\boldsymbol{x}, \tau_i))) + 0 = 1 + 0 = 1$.

□

We then use induction based on the TAC C's topological structure.

Lemma 2. *Let $\boldsymbol{\theta}$ be a well-specified parameter vector for* TAC $C = (\mathbf{Z}, \mathbf{E})$. *The conditional partition function* $\mathcal{Z}_{\boldsymbol{\theta}}(\mathbf{z}_1) = 1$.

Proof. We use induction on the number of nodes k, following the topological sort $\mathbf{z}_1, \dots \mathbf{z}_M$. For clarity, here we abuse the subscript notation for topological order, and therefore $\mathbf{z}_M$ (instead of $\mathbf{z}_2$) is the output.

Let C_k be the sub-TAC induced by $\{\mathbf{z}_1, \dots, \mathbf{z}_k\}$. Its partition function is $\mathcal{Z}_k(\mathbf{z}_1) = \sum_{\mathbf{z}_2 \dots \mathbf{z}_k} \prod_{m=2}^{k} \tilde{p}_{\boldsymbol{\theta}}(\mathbf{z}_m \mid S_m)$, where S_m denotes the source nodes of $\mathbf{z}_m$ under its corresponding hyperedge.

Base Case. $k = 1$. C_1 has only $\mathbf{z}_1$. $\mathcal{Z}_1(\mathbf{z}_1) = 1$ since the product is empty.

Inductive Step. We assume $\mathcal{Z}_{k-1}(\mathbf{z}_1) = 1$. First we rewrite $\mathcal{Z}_k(\mathbf{z}_1)$ by explicitly summing over $\mathbf{z}_k$. Since $\mathbf{z}_1, \dots \mathbf{z}_k$ is a topological order, the source nodes of $\mathbf{z}_k$: S_k is a subset of $\{\mathbf{z}_1, \dots \mathbf{z}_{k-1}\}$. We thus rewrite $\mathcal{Z}_k(\mathbf{z}_1)$ as

$$\mathcal{Z}_k(\mathbf{z}_1) = \sum_{\mathbf{z}_2 \dots \mathbf{z}_k} \left(\prod_{m=2}^{k-1} \tilde{p}_\theta(\mathbf{z}_m \mid S_m) \right) \cdot \left(\sum_{\mathbf{z}_k} \tilde{p}_\theta(\mathbf{z}_k \mid S_k) \right). \tag{7}$$

We discuss the summands by the validity of $\mathbf{z}_2 \dots \mathbf{z}_{k-1}$:

- If $\mathbf{z}_2 \dots \mathbf{z}_{k-1}$ is valid: by Lemma 1 the term $\sum_{\mathbf{z}_k} \tilde{p}_{\boldsymbol{\theta}}(\mathbf{z}_k \mid S_k) = 1$. This summand is therefore $\prod_{m=2}^{k-1} \tilde{p}_{\boldsymbol{\theta}}(\mathbf{z}_m \mid S_m)$.
- If $\mathbf{z}_2 \dots \mathbf{z}_{k-1}$ is not valid: by Eqs 1 and 2 this summand is 0.

We can thus rewrite Eq. (7) as

$$\mathcal{Z}_k(\mathbf{z}_1) = \sum_{\mathbf{z}_2, \dots, \mathbf{z}_{k-1} \mid \text{valid assignments}} \prod_{m=2}^{k-1} \tilde{p}_{\boldsymbol{\theta}}(\mathbf{z}_m \mid S_m). \tag{8}$$

Equation (8) can be further rewritten to sum over both valid and invalid $\mathbf{z}_2, \dots, \mathbf{z}_{k-1}$ assignments (since again by Eqs. (1) and (2), the summand is 0 for invalid assignments):

$$\mathcal{Z}_k(\mathbf{z}_1) = \sum_{\mathbf{z}_2, \dots, \mathbf{z}_{k-1}} \prod_{m=2}^{k-1} \tilde{p}_{\boldsymbol{\theta}}(\mathbf{z}_m \mid S_m) = \mathcal{Z}_{k-1}(\mathbf{z}_1). \tag{9}$$

Since by assumption $\mathcal{Z}_{k-1}(\mathbf{z}_1) = 1$, we thus prove by induction $\mathcal{Z}_M(\mathbf{z}_1) = \mathcal{Z}_{\boldsymbol{\theta}}(\mathbf{z}_1) = 1$. □

Finally, we show that Lemma 2 implies the equivalence of maximizing the normalized and unnormalized likelihoods when the true parameters are well-specified.

Theorem 1. *Let $\boldsymbol{\Theta}$ be the entire parameter space and let $\boldsymbol{\Theta}' \subseteq \boldsymbol{\Theta}$ be the subset of well-specified parameters. Assume $\boldsymbol{\theta}^*$ uniquely maximizes the normalized likelihood $p_{\boldsymbol{\theta}}(\mathbf{z}_{2..M}|\mathbf{z}_1)$ and resides $\in \boldsymbol{\Theta}'$. Then, $\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \tilde{p}_{\boldsymbol{\theta}}(\mathbf{z}_{2..M}|\mathbf{z}_1) \implies \hat{\boldsymbol{\theta}} = \boldsymbol{\theta}^*$.*

Proof. First we note $\forall \boldsymbol{\theta} \in \boldsymbol{\Theta}, \mathcal{Z}_{\boldsymbol{\theta}}(\mathbf{z}_1) \leq 1$, since for any adaptor $\sum_y \tilde{p}_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}) \leq 1$. By Eqs. (1) and (2) the global partition function must also be ≤ 1.

We rewrite the unnormalized likelihood as a product of normalized likelihood and the partition function:

$$\tilde{p}_{\boldsymbol{\theta}}(\mathbf{z}_{2 \dots M} \mid \mathbf{z}_1) = p_{\boldsymbol{\theta}}(\mathbf{z}_{2 \dots M} \mid \mathbf{z}_1) \cdot \mathcal{Z}_{\boldsymbol{\theta}}(\mathbf{z}_1) \tag{10}$$

Since $\mathcal{Z}_{\boldsymbol{\theta}}(\mathbf{z}_1) \leq 1, \forall \boldsymbol{\theta} \in \boldsymbol{\Theta}, \tilde{p}_{\boldsymbol{\theta}}(\mathbf{z}_{2 \dots M} \mid \mathbf{z}_1) \leq p_{\boldsymbol{\theta}}(\mathbf{z}_{2 \dots M} \mid \mathbf{z}_1)$.

At the well-specified true parameters $\boldsymbol{\theta}^*$ we have $\mathcal{Z}_{\boldsymbol{\theta}}(\mathbf{z}_1) = 1$ by Lemma 2. Therefore $\tilde{p}_{\boldsymbol{\theta}^*}(\mathbf{z}_{2 \dots M} \mid \mathbf{z}_1) = p_{\boldsymbol{\theta}^*}(\mathbf{z}_{2 \dots M} \mid \mathbf{z}_1)$.

By our assumption that $\boldsymbol{\theta}^*$ maximizes normalized likelihood, $\forall \boldsymbol{\theta} \in \boldsymbol{\Theta}, p_{\boldsymbol{\theta}^*}(\mathbf{z}_{2 \dots M} \mid \mathbf{z}_1) \geq p_{\boldsymbol{\theta}}(\mathbf{z}_{2 \dots M} \mid \mathbf{z}_1)$.

Combining everything together:

$$\begin{aligned}\tilde{p}_{\boldsymbol{\theta}^*}(\mathbf{z}_{2\ldots M} \mid \mathbf{z}_1) &= p_{\boldsymbol{\theta}^*}(\mathbf{z}_{2\ldots M} \mid \mathbf{z}_1) \\ &\geq p_{\boldsymbol{\theta}}(\mathbf{z}_{2\ldots M} \mid \mathbf{z}_1) \\ &\geq \tilde{p}_{\boldsymbol{\theta}}(\mathbf{z}_{2\ldots M} \mid \mathbf{z}_1)\end{aligned}$$

for all $\boldsymbol{\theta} \in \boldsymbol{\Theta}$. Under the assumption $\boldsymbol{\theta}^*$ is unique, $\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \tilde{p}_{\boldsymbol{\theta}}(\mathbf{z}_{2\ldots M} \mid \mathbf{z}_1) = \hat{\boldsymbol{\theta}}$. □

Theorem 2. *Let $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1 \ldots \boldsymbol{\theta}_K\}$ be the union of a K-adaptor* TAC*'s LM adaptor parameters . If $\forall \mathbf{z}_{k,1} \in \Sigma^*, \mathbf{z}_{k,2} \in \Sigma^*, \|\nabla_{\boldsymbol{\theta}} \left(\sum \log p_{LM}(\mathbf{z}_{k,2} \mid \mathbf{z}_{k,1}; \boldsymbol{\theta})\right)\|_\infty \leq G$, then $\nabla_{\boldsymbol{\theta}} \log \mathcal{Z}_{\boldsymbol{\theta}} \leq 2G(1 - \mathcal{Z}_{\boldsymbol{\theta}})$.*

Proof. Here we fix $\mathbf{z}_1 = x$. We denote $\mathbf{z}_{2\ldots M} = y$. Let $p^{(k)}_{LM}(y)$ be the k-th LM adaptor's unmasked node probability, given (x, y) as TAC input and output. We then denote $p_{\boldsymbol{\theta}}(y) = \prod_k p^{(k)}_{LM}$ as a TAC's *normalized* distribution over node assignments (without masking invalid ones). The partition function $\mathcal{Z}_{\boldsymbol{\theta}} = \sum_y p_{\boldsymbol{\theta}}(y \mid x)\mathbb{I}(y \in V) = \Pr_{p_{\boldsymbol{\theta}}}(V)$ where V is the set of valid node assignments.

We first rewrite $\nabla_{\boldsymbol{\theta}} \log \mathcal{Z}_{\boldsymbol{\theta}}$ as an expectation under $p_{\boldsymbol{\theta}}$:

$$\nabla_{\boldsymbol{\theta}} \log \mathcal{Z}_{\boldsymbol{\theta}} = \mathbb{E}_{y\sim p_{\boldsymbol{\theta}}(\cdot|V)} \left[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(y)\right]. \tag{11}$$

Using the identity $\sum_y p_{\boldsymbol{\theta}}(y)\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(y) = 0$, we rewrite Eq. (11) as

$$\nabla_{\boldsymbol{\theta}} \log \mathcal{Z}_{\boldsymbol{\theta}} = \mathbb{E}_{y\sim p_{\boldsymbol{\theta}}(\cdot|V)} \left[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(y)\right] - \mathbb{E}_{y\sim p_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(y)\right]. \tag{12}$$

Let $f = \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(y)$. We can now rewrite $\|\nabla_{\boldsymbol{\theta}} \log \mathcal{Z}_{\boldsymbol{\theta}}\|_\infty$ as

$$\begin{aligned}\|\nabla_{\boldsymbol{\theta}} \log \mathcal{Z}_{\boldsymbol{\theta}}\|_\infty &= \|\mathbb{E}_{p_{\cdot|V}} [f] - \mathbb{E}_{p_{\boldsymbol{\theta}}} [f]\|_\infty \\ &= \|\sum_y f \cdot (p_{\boldsymbol{\theta}}(y \mid V) - p_{\boldsymbol{\theta}}(y))\|_\infty \\ &\leq \sum_y \|f\|_\infty \cdot |p_{\boldsymbol{\theta}}(y \mid V) - p_{\boldsymbol{\theta}}(y)| \\ &\leq \sum_y G \cdot |p_{\boldsymbol{\theta}}(y \mid V) - p_{\boldsymbol{\theta}}(y)|. \end{aligned} \tag{13}$$

Noting that $\sum_y |p_{\boldsymbol{\theta}}(y \mid V) - p_{\boldsymbol{\theta}}(y)|$ is twice the total variation between $p_{\boldsymbol{\theta}}$ and $p_{\boldsymbol{\theta}}(\cdot \mid V)$, and that the total variation between $p_{\boldsymbol{\theta}}$ and $p_{\boldsymbol{\theta}}(\cdot \mid V)$ is $(1 - \mathcal{Z}_{\boldsymbol{\theta}})$ — the sum of invalid assignments' probabilities under $p_{\boldsymbol{\theta}}$ — we can rewrite Eq. (13) as $\|\nabla_{\boldsymbol{\theta}} \log \mathcal{Z}_{\boldsymbol{\theta}}\|_\infty \leq 2G(1 - \mathcal{Z}_{\boldsymbol{\theta}})$.

□

E Implementation Considerations

In this section we discuss practical implementation considerations. In particular, we distinguish between *one-time* and *per-use* efforts.

E.1 One-Time Efforts

Parsing and canonicalization. There exist multiple libraries that can readily be used to implement `parse` and `canon` for typed data-holding objects in Python. One example is LangFun which we use extensively in the paper. Another popular library is Pydantic, which is used in DSPy.

Type validation logic. As we briefly discussed in Footnote 5, the `parse` function can be used to implement complex business logic. Such logic can usually be implemented cleanly as part of type definition (*e.g.*, as `__init__` and `__post_init__` methods in Python).

Algorithms. The core TAC algorithms for execution and training (Algorithms listed in §C) are general and need only be implemented once. The main computational bottlenecks in these algorithms are:

- Sampling from an LM adaptor $p_{LM}(\cdot; \boldsymbol{\theta})$.
- Evaluating the conditional probability of $\boldsymbol{y}$ given $\boldsymbol{x}$ under an LM adaptor: $p_{LM}(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$.
- Computing gradients of $(\boldsymbol{x}, \boldsymbol{y})$ with regard to parameters $\boldsymbol{\theta}$: $\nabla_{\boldsymbol{\theta}} \log p_{LM}(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$.

A practical implementation can abstract these bottlenecks away, by offloading these intensive parts to dedicated inference servers (e.g., vLLM). The core TAC logic remains a lightweight, accelerator-agnostic program. Furthermore, since TACs use parameter-efficient fine-tuning (PEFT), the adaptor weights and gradients are small enough to be processed quickly, often without needing dedicated accelerators for the logic itself. This design significantly reduces the low-level engineering burden.

E.2 Per-Use Efforts

Once the core engine is in place, a practitioner's effort is focused on defining a TAC hypergraph for their specific task. Since the TAC hypergraph is essentially a data flow graph, it can be represented in a way that is directly analogous to network architecture definitions in popular neural network frameworks such as PyTorch, where the `Module` s represent hyperedges, and their `forward` methods connect the typed data nodes.

F Additional TAC Diagrams of Trainable Workflows

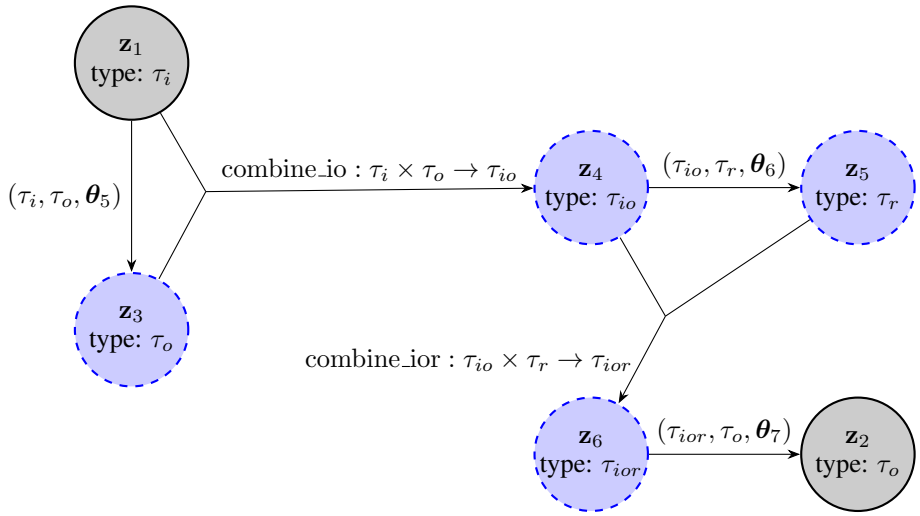


Figure 6: **refine-structure**: refinement through cascade topology engineering. This cascade models a refinement process where an initial output sketch is iteratively refined based on generated rationales.

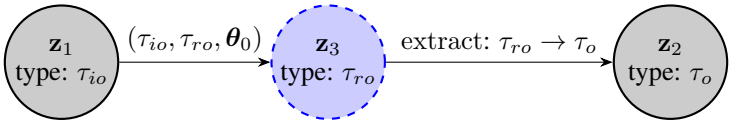


Figure 7: C_{fallback} for **cot-type-structure**. Notice that the adaptor $(\tau_{io}, \tau_{ro}, \boldsymbol{\theta}_0)$ uses 'fallback' weights $\boldsymbol{\theta}_0$ that represent no-op weights. Since we conduct experiment on LoRA adaptors in this work, we use the zero-init vectors as $\boldsymbol{\theta}_0$.

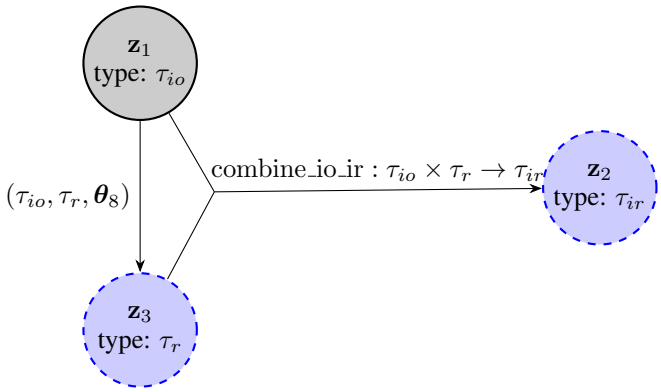


Figure 8: Inference network TAC C' for **cot-type-structure**.

G Further Details of Experiment Setup

Data splits. We focus on the low-data regime of task adaptation in this work. For MGSM and MGSM-SymPy, each language has 100/30/120 training/validation/test examples respectively. The splits are 100/30/100 and 100/30/300 for HotPotQA and FinQA respectively. For HotPotQA and FinQA, we use the first entries from the original dataset files as our training and evaluation subsets. For MGSM experiments, we train and evaluate on each language separately. For MuSR tasks, the splits are $100/30/120$ and $100/30/126$ respectively.

Evaluation. We look at exact match accuracy scores of the answers for all 5 tasks. For MGSM-SymPy experiments, we convert answers from the dataset to integers; as for the model predictions, we evaluate the expressions as rational numbers under SymPy[16], and cast the results as integer numbers. We do not make use of additional clues from the datasets (*e.g.*, the rationales provided for the 8 examples in MGSM datasets).

G.1 TAC Setup

Training procedure. We train all workflows that have latent variables with our TACSTaR and Amortized TACSTaR algorithms, except for the original (untyped) STaR experiments. Since **direct** experiments do not have latent variables, we train those models using the ordinary cross entropy loss. In all experiments we use a batch size of 8. The Adam optimizer (Kingma & Ba, 2014) is used throughout all experiments, with a learning rate of $5e-5$. We early-stop if no higher validation score is achieved for 4 consecutive epochs. The sampler configuration κ is set to use a combination of top-K and nucleus sampling (Holtzman et al., 2020), where we first choose the top 40 candidates, and cut off accumulated probability mass at 0.95. To train the inference TACs, we accumulate 32 samples from C_{infer} and 16 samples from the fallback model (that is, $G = 48$ at the end of Algorithm 5).

[16] `https://www.sympy.org/en/index.html`

Decoding procedure for generation tasks. Here we denote the answer type as τ_o. For each test input instance, we obtain 32 samples $\hat{\mathbf{Z}}^{(1)} \dots \hat{\mathbf{Z}}^{(32)}$ using `forward`, bucket their output node values $\texttt{parse}(\hat{\mathbf{z}}_2^{(1)}, \tau_o) \dots \texttt{parse}(\hat{\mathbf{z}}_2^{(32)}, \tau_o)$ into B bins, identified by the parsed output $y_1 \dots y_B$. We output the answer with maximum accumulated unnormalized probability mass, namely $\arg\max_b \sum_{s \in [1..32], \text{parse}(\hat{\mathbf{z}}_2, \tau_o) = y_b} \tilde{p}_{\boldsymbol{\theta}}(\hat{\mathbf{Z}}^{(s)})$.

Decoding procedure for classification tasks. We estimate each label c's normalized marginal probability using Eq. (4), with $N = 32$. We output the label with largest normalized marginal probability as prediction.

Object representation of data. We represent input τ_i and output τ_o as Python types. The objects are encoded as string representations under LangFun. We design the input and output types separately to reflect the original dataset schemata (Listings 1 to 3). As for the rationales (represented by τ_r in **cot-type-structure** and **cot-cascade-structure**) we represent them as lists of strings (Listing 4). Product types are represented as new Python classes (*e.g.*, the product of type `Question` and `Answer`, represented as τ_{io} in Figs. 7 and 8, is a new class QuestionAnswer). The object representation can be arbitrarily complex, with LangFun handling all `canon` and `parse` logic (for example, Listing 6 has `Answer` objects embedded in multiple types; and Listing 7 has self-referential definitions).

```
class Question:
  question: str


class Answer:
  answer: str
```

Listing 1: Input and output type definitions for MGSM

```
class Paragraph:
  title: str
  sentences: list[str]


class Context:
  paragraphs: list[Paragraph]



class Answer:
  answer: str


class Question:
  id: str
  question: str
  context: Context
```

Listing 2: Input and output type definitions for HotPotQA

```
class Question:
  question: str
  pre_text: list[str]
  table: list[list[str]]
```

```
  post_text: list[str]


class Step:
  op: str
  arg1: str
  arg2: str
  res: str


class Answer:
  answer: str


class QuestionAnswer:
  question: Question
  answer: Answer


class Answer:
  answer: str
```

Listing 3: Input and output type definitions for FinQA

```
class Rationale:
  steps: list[str]
```

Listing 4: `Rationale` type definition

```
class QuestionAnswer:
  question: Question
  answer: Answer
```

Listing 5: `QuestionAnswer` type definition

```
class ThinkingSteps:
  steps: list[str]


class Paragraph:
  title: str
  sentences: list[str]


class Context:
  paragraphs: list[Paragraph]


class SupportingFact:
  title: str
  sentence: str


class RelevantContext:
```

```
  sentences: list[str]


class Answer:
  answer: str


class Question:
  id: str
  question: str
  context: Context


class QuestionAnswer:
  question: Question
  answer: Answer


class AnswerFirstAttemptThinkingStepsAnswer:
  answer_first_attempt: Answer
  thinking_steps: ThinkingSteps
  answer: Answer


class QuestionAnswerFirstAttempt:
  question: Question
  answer_first_attempt: Answer


class QuestionAnswerFirstAttemptThinkingSteps:
  question: Question
  answer_first_attempt: Answer
  thinking_steps: ThinkingSteps
```

Listing 6: Type definitions for **refine-structure** on HotPotQA

```
class Expression:
  operator: Literal['+', '-', '*', '/']
  left: Union[int, 'Expression']
  right: Union[int, 'Expression']

class Answer:
  answer: Expression
```

Listing 7: `Expression` type definitions in MGSM **expression-cascade-structure** experiments

G.2 DSPy SETUP

We conduct most of the DSPy experiments under v 3.0.1, but report results from DSPy v 2.6.19 for `gemini-1.1-7b-it` experiments since both BFSWRS and MIPROv2 struggle to generate valid outputs under DSPy v 3.0.1. Moreover, the non-optimized MGSM average accuracy is much lower under v 3.0.1 (for Native CoT it is 0.7% under v 2.6.19, and 0.2% under v 3.0.1). For all other experiments, we report results from DSPy v 3.0.1 which sets up JSON schema-based constrained decoding correctly out-of-the-box. As we noted in §4.2, constrained decoding significantly improves performance for tasks with structured output.

We serve base models on vLLM v 0.10.0.

Input and output object definitions. For structured input and output tasks, we subclass `dspy.Signature` as `QASignature` to represent examples. The property names and types in a `QASignature` class are identical to counterparts in TAC experiments. FinQA and MGSM-SymPy signatures are listed in Listing 8 and Listing 9 respectively.

```
class QASignature(dspy.Signature):
  pre_text: list[str] = dspy.InputField()
  table: list[list[str]] = dspy.InputField()
  post_text: list[str] = dspy.InputField()
  question: str = dspy.InputField()
  answer: str = dspy.OutputField()
```

Listing 8: DSPy object signature for FinQA. Property names and types are identical to their TAC counterparts in Listing 3

```
class Expression(pydantic.BaseModel):
  operator: Literal['+', '-', '*', '/']
  left: Union[int, float, 'Expression']
  right: Union[int, float, 'Expression']

class QASignature(dspy.Signature):
  question: str = dspy.InputField()
  answer: Expression = dspy.OutputField()
```

Listing 9: DSPy object signature for MGSM-SymPy. Property names and types are identical to their TAC counterparts in Listing 7

DSPy models. We conduct reasoning experiments on both the native `dspy.ChainOfThought` module, and an explicitly two-step composite module that resembles TAC **cot-cascade-structure** patterns. Two-step modules for FinQA and MuSR are listed in Listings 10 and 11 as examples.

```
class QuestionRationale(dspy.Signature):
  question: str = dspy.InputField()
  pre_text: list[str] = dspy.InputField()
  table: list[list[str]] = dspy.InputField()
  post_text: list[str] = dspy.InputField()
  question: str = dspy.InputField()
  rationale: list[str] = dspy.OutputField()

class RationaleAnswer(dspy.Signature):
  rationale: list[str] = dspy.InputField()
  answer: str = dspy.OutputField()

class TwoStepPredictor(dspy.Module):
  def __init__(self):
    self.question_to_rationale = dspy.Predict(QuestionRationale)
    self.rationale_to_answer = dspy.Predict(RationaleAnswer)

  def forward(self, pre_text: list[str], table: list[list[str]], post_text:
      list[str], question: str):
    r = self.question_to_rationale(question=question, pre_text=pre_text, table=
        table, post_text=post_text).rationale
```

```
    return dspy.Prediction(answer=self.rationale_to_answer(rationale=r).answer)
```

Listing 10: DSPy two-step reasoning model definition for FinQA

```
class QuestionRationale(dspy.Signature):
  context: str = dspy.InputField()
  question: str = dspy.InputField()
  choices: list[str] = dspy.InputField()
  rationale: list[str] = dspy.OutputField()

class RationaleAnswer(dspy.Signature):
  rationale: list[str] = dspy.InputField()
  choices: list[str] = dspy.InputField()
  answer: str = dspy.OutputField()

class TwoStepPredictor(dspy.Module):
  def __init__(self):
    self.question_to_rationale = dspy.Predict(QuestionRationale)
    self.rationale_to_answer = dspy.Predict(RationaleAnswer)

  def forward(self, context: str, question: str, choices: list[str]):
    r = self.question_to_rationale(question=question, context=context, choices=
        choices).rationale
    return dspy.Prediction(answer=self.rationale_to_answer(rationale=r, choices
        =choices).answer)
```

Listing 11: DSPy two-step reasoning model definition for MuSR

Prompt optimization under DSPy. We experiment with optimizers `dspy.MIPROv2` and `dspy.BootstrapFewShotWithRandomSearch` (listed as BFSWRS below). For MGSM-SymPy and FinQA experiments we do not report BFSWRS results, as they consistently need more context length than the model maximum (8192). Moreover, for FinQA experiments we resort to MIPROv2 0-shot due to similar context length problems.

We set `max_errors=2` for all optimizers. For MiPROv2 we set `auto='medium'`. For MiPROv2 with 0-shot settings we additionally set `max_bootstrapped_demos=0, max_labed_demos=0`.

H PER-LANGUAGE TAC AND ORIGINAL STAR MGSM AND MGSM-SYMPY RESULTS

Per-language TAC and original STaR experimental results on tasks MGSM and MGSM-SymPy are listed in Tables 5 and 6.

Pattern	Adaptation Method	es	en	de	fr	zh	ru	ja	te	th	Average
direct	TACSTaR	27.5	27.5	25.0	25.0	23.3	25.8	23.3	18.3	26.7	24.7
cot-type-structure	TACSTaR	80.0	84.2	76.7	83.3	80.0	85.0	71.7	79.2	83.3	80.4
cot-cascade-structure	TACSTaR	87.5	87.5	83.3	85.8	80.0	87.5	74.2	73.3	80.8	82.2
refine-structure	TACSTaR	86.7	90.0	76.7	77.5	73.3	78.3	69.2	72.5	83.3	78.6
expression-cascade-structure	TACSTaR	83.3	82.5	83.3	75.8	70.0	79.2	65.8	75.0	75.8	75.9
cot-cascade-structure	un-adapted	42.5	47.5	46.7	42.5	45.0	53.3	31.7	45.0	54.2	45.4
cot-type-structure	un-adapted	77.5	79.2	80.8	76.7	68.3	79.2	68.3	69.2	73.3	74.7
expression-cascade-structure	un-adapted	76.7	71.7	69.2	70.8	68.3	68.3	63.3	70.8	73.3	69.5
cot-cascade-structure	amortized TACSTaR	84.2	91.7	86.7	83.3	82.5	81.7	70.8	77.5	83.3	82.4
N/A	original STaR	74.2	79.2	75.8	75.8	70.0	88.3	74.2	75.8	75.8	76.9

Table 5: `gemma-2-27b-it` MGSM and MGSM-SymPy per-language accuracies (TAC and original STaR experiments).

Pattern	Adaptation Method	es	en	de	fr	zh	ru	ja	te	th	Average
direct	TACSTaR	5.8	6.7	6.7	8.3	7.5	2.5	5.0	1.7	1.7	5.1
cot-cascade-structure	TACSTaR	40.8	35.8	31.7	29.2	24.2	31.7	13.3	18.3	20.8	27.3
cot-cascade-structure	un-adapted	$8.0 \cdot 10^{-1}$	0.0	$8.0 \cdot 10^{-1}$	0.0	0.0	$8.0 \cdot 10^{-1}$	0.0	1.7	0.0	$5.0 \cdot 10^{-1}$
N/A	original STaR	15.0	27.5	1.7	5.8	22.5	0.0	3.3	9.2	9.2	10.5

Table 6: `gemma-1.1-7b-it` MGSM per-language accuracies (TAC and original STaR experiments).

I PER-TASK TAC MUSR RESULTS

Per-task TAC experimental results on task MuSR are listed in Tables 7 and 8.

Decoding Method	Murder Mystery	Object Placements	Team Allocation	Average
Generation	61.7	51.6	41.7	51.6
Classification	65.0	50.0	80.0	65.0

Table 7: `gemma-2-27b-it` MuSR per-task accuracies (TAC experiments).

Decoding Method	Murder Mystery	Object Placements	Team Allocation	Average
Generation	60.0	43.7	82.5	62.1
Classification	59.2	42.9	85.8	62.6

Table 8: `gemma-1.1-7b-it` MuSR per-task accuracies (TAC experiments).

J PER-TASK DSPY MUSR RESULTS

Per-task DSPy experimental results on task MuSR are listed in Tables 9 and 10.

Model	Optimizer	Murder Mystery	Object Placements	Team Allocation	Average
Native CoT	None	20.8	0.0	0.0	6.9
Native CoT	MIPRO 0-shot	40.8	$7.9 \cdot 10^{-1}$	0.0	13.9
Native CoT	MIPRO	51.7	50.8	49.2	50.5
Two-step	None	52.5	14.3	22.5	29.8
Two-step	MIPRO 0-shot	55.0	27.8	19.2	34.0
Two-step	MIPRO	59.2	44.4	50.8	51.5

Table 9: `gemma-2-27b-it` MuSR per-task accuracies (DSPy experiments).

Model	Optimizer	Murder Mystery	Object Placements	Team Allocation	Average
Native CoT	None	10.0	3.2	3.3	5.5
Native CoT	MIPRO 0-shot	6.7	3.2	2.5	4.1
Native CoT	MIPRO	34.2	25.4	50.0	36.5
Two-step	None	33.3	5.6	16.7	18.5
Two-step	MIPRO 0-shot	35.8	1.6	15.0	17.5
Two-step	MIPRO	44.2	32.5	26.7	34.5

Table 10: `gemma-1.1-7b-it` MuSR per-task accuracies (DSPy experiments).

Model	Optimizer	Murder Mystery	Object Placements	Team Allocation	Average
Native CoT	None	0.0	0.0	0.0	0.0
Native CoT	MIPRO 0-shot	0.0	0.0	0.0	0.0
Native CoT	MIPRO	55.8	50.8	47.5	51.4
Two-step	None	4.2	$7.9 \cdot 10^{-1}$	0.0	1.7
Two-step	MIPRO 0-shot	3.3	1.6	0.0	1.6
Two-step	MIPRO	65.0	59.5	60.0	61.5

Table 11: `Qwen3-8B` MuSR per-task accuracies (DSPy experiments).

K PER-LANGUAGE DSPY MGSM AND MGSM-SYMPY RESULTS

Per-language DSPy experimental results on tasks MGSM and MGSM-SymPy are listed in Tables 12 to 14.

Model	Optimizer	es	en	de	fr	zh	ru	ja	te	th	Average
Native CoT	None	55.0	57.5	52.5	51.7	54.2	59.2	45.0	39.2	40.0	50.5
Native CoT	BFSWRS	84.2	89.2	87.5	81.7	75.0	87.5	75.0	77.5	79.2	81.9
Native CoT	MIPROv2	82.5	86.7	81.7	76.7	77.5	84.2	70.0	74.2	75.8	78.8
Two-step	None	1.7	5.8	2.5	1.7	3.3	1.7	1.7	3.3	5.0	3.0
Two-step	MIPROv2	76.7	83.3	76.7	78.3	73.3	79.2	70.0	67.5	71.7	75.2
Two-step	BFSWRS	80.8	84.2	76.7	81.7	70.0	81.7	67.5	64.2	72.5	75.5

Table 12: `gemma-2-27b-it` MGSM per-language accuracies (DSPy experiments).

Model	Optimizer	es	en	de	fr	zh	ru	ja	te	th	Average
Native CoT	None	$8.0 \cdot 10^{-1}$	$8.0 \cdot 10^{-1}$	$8.0 \cdot 10^{-1}$	0.0	0.0	2.5	1.7	0.0	0.0	$7.0 \cdot 10^{-1}$
Native CoT	BFSWRS	0.0	$8.0 \cdot 10^{-1}$	1.7	5.0	$8.0 \cdot 10^{-1}$	1.7	1.7	2.5	0.0	1.6
Native CoT	MIPROv2	$8.0 \cdot 10^{-1}$	1.7	2.5	2.5	1.7	0.0	1.7	$8.0 \cdot 10^{-1}$	$8.0 \cdot 10^{-1}$	1.4
Two-step	None	0.0	0.0	$8.0 \cdot 10^{-1}$	0.0	0.0	0.0	1.7	0.0	0.0	$3.0 \cdot 10^{-1}$
Two-step	MIPROv2	0.0	0.0	0.0	0.0	0.0	$8.0 \cdot 10^{-1}$	0.0	0.0	$8.0 \cdot 10^{-1}$	$2.0 \cdot 10^{-1}$
Two-step	BFSWRS	0.0	0.0	0.0	0.0	0.0	$8.0 \cdot 10^{-1}$	0.0	0.0	$8.0 \cdot 10^{-1}$	$2.0 \cdot 10^{-1}$

Table 13: `gemma-1.1-7b-it` MGSM per-language accuracies (DSPy experiments).

Model	Optimizer	es	en	de	fr	zh	ru	ja	te	th	Average
Native CoT	None	56.7	66.7	55.0	45.8	47.5	59.2	45.0	49.2	45.8	52.3
Native CoT	MIPROv2	66.7	64.2	58.3	60.8	56.7	62.5	50.8	42.5	51.7	57.1
Two-step	None	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Two-step	MIPROv2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 14: `gemma-2-27b-it` MGSM-SymPy per-language accuracies (DSPy experiments).

L DSPy FinQA results

DSPy experimental results on the FinQA task are listed in Table 15 and Table 16.

Model	Optimizer	Accuracy
Native CoT	None	11.7
Native CoT	MIPROv2 0-shot	12.7
Two-step	None	5.7
Two-step	MIPROv2 0-shot	10.7

Table 15: `gemma-2-27b-it` FinQA accuracy (DSPy experiments).

Model	Optimizer	Accuracy
Native CoT	None	0.0
Native CoT	MIPROv2 0-shot	$6.7 \cdot 10^{-1}$
Two-step	None	0.0
Two-step	MIPROv2 0-shot	$3.3 \cdot 10^{-1}$

Table 16: `gemma-1.1-7b-it` FinQA accuracy (DSPy experiments).

Model	Optimizer	Accuracy
Native CoT	None	4.3
Native CoT	MIPROv2 0-shot	5.3
Two-step	None	1.0
Two-step	MIPROv2 0-shot	12.0

Table 17: `Qwen3-8B` FinQA accuracy (DSPy experiments).

M EXAMPLE EXPRESSIONS FROM **expression-cascade-structure** UNDER THE MGSM-SYMPY TASK

See Table 18.

Question	Answer	Expression
Nissa hires 60 seasonal workers to play elves in her department store's Santa village. A third of the elves quit after children vomit on them, then 10 of the remaining elves quit after kids kick their shins. How many elves are left?	20	$(60 - (60/3)) - 10$
The expenditure of Joseph in May was $500. In June, his expenditure was $60 less. How much was his total expenditure for those two months?	940	$500 + 440$
Tom gets 4 car washes a month. If each car wash costs $15 how much does he pay in a year?	720	$(15 \times 4) \times 12$

Table 18: Example arithmetic expressions generated for MGSM questions by **expression-cascade-structure**.

N EXAMPLE INSTRUCTION PROMPT GENERATED BY LANGFUN

The LangFun library translates requests that transformed a typed object into another typed object into natural language instructions for LLMs, to facilitate its `parse` operations. For example, Listing 12 is a prompt generated by LangFun for the request that transforms a `Question` object into an Answer object.

````
Please respond to the last INPUT_OBJECT with OUTPUT_OBJECT according to
    OUTPUT_TYPE.

INPUT_OBJECT:
  1 + 1 =

OUTPUT_TYPE:
  Answer

  ```python
  class Answer:
    final_answer: int
  ```

OUTPUT_OBJECT:
  ```python
  Answer(
    final_answer=2
  )
  ```

INPUT_OBJECT:
  ```python
  Question(
    question='How are you?'
  )
  ```

OUTPUT_TYPE:
````

````
  Answer

  ```python
  class Answer:
    answer: str
  ```

OUTPUT_OBJECT:
````

Listing 12: Example instruction prompt generated by LangFun