LEARNING ERROR PATTERNS OF LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

When trying to get to LLMs generate outputs satisfying a validity constraint, grammars capture only part of the constraint, while fine-tuning is costly and may collapse output diversity in sparse domains. We observe that a given LLM on a given domain fails in a small number of recurring ways: for example, by repeatedly using Python function names when generating TypeScript. We capture these error patterns with *prefix filters*: short, interpretable Python predicates, specific to a model and a domain, that flag whether a partial output will fail. Our algorithm, PALLA, learns these filters from a few hundred samples using the domain's oracle (e.g., a compiler) to flag failures and an external LLM to write the filters, at a cost of roughly $5–$15 per domain. Across 4 domains and 7 LLMs, the learned filters show that errors are clustered, with three filters catching half of Qwen2.5-7B's MLIR errors, and largely model-specific, rarely transferring between models except where every model fails in the same way. Used with constrained adaptive rejection sampling, the filters raise validity without retraining: Qwen2.5-14B's MLIR compile rate rises from 27.5% to 71.0%, where grammar-constrained decoding lowers it, and Qwen2.5-1.5B's TypeScript compile rate rises from 16.0% to 70.0%, above that of unconstrained Llama3.1-8B. Unlike fine-tuning on the same samples, which collapses output diversity, the filters remove only the outputs they flag.

1 INTRODUCTION

When using large language models (LLMs), for many domains there exists a clear set of *bad outputs*, of *failures*, we want the LLM to avoid. Modern LLMs are imperfect and sometimes generate these bad outputs: for example, or a model generating code may produce programs that fail to compile. Techniques such as grammar-constrained decoding (Geng et al., 2023) help, but are imperfect as the notion of failures (e.g., programs that are not type-safe) often cannot be precisely captured via a grammar. Fine-tuning on the model's mistakes can also help, but needs more data and compute, and can collapse the diversity of the model's outputs.

A key observation is that although the space of failures can be complex, in practice the errors that LLMs make are not uniformly distributed across this space. Instead, a given model on a given domain tends to fail in a small number of systematic and repetitive ways. For example, when generating functions in the MLIR language (Lattner et al., 2021) (a low-resource language used to write compilers), Qwen2.5-7B (Yang et al., 2025b) frequently produces programs containing the nonexistent operator `torch.aten.softmax`.; or, Llama3.1-8B (Grattafiori et al., 2024) often inserts the character `%` in invalid locations. This observation motivates the central question on controlling model behavior we ask in this paper:

Can we learn and suppress the error patterns that a given LLM actually exhibits on a given domain?

Prefix filters. This paper answers the above question by proposing *prefix filters*: per-domain-and-LLM *symbolic* functions (e.g., Python functions) that analyze the prefix of a sample being generated, and decide whether the sample has already gone wrong or is likely to do so in the future. For example, an MLIR sample prefix containing the undefined `torch.aten.softmax` operator will fail to compile regardless of completion—the goal of prefix filters is to identify such samples early.

Prefix filters are a solution to our central question in two ways: First, they are *interpretable* functions that capture the error pattern of an LLM. Second, prefix filters can be used directly to prevent failing samples, i.e., by rejecting prefixes caught by the filters and resampling (Parys et al., 2025).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

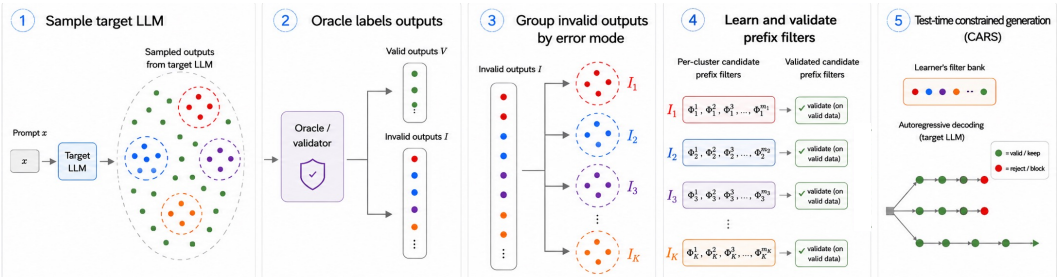


Figure 1: Overall pipeline of PALLA: we exploit the fact that error modes are clustered to learn prefix filters targeting each cluster, then use the filters to avoid samples from the clustered errors at test time.

Constructing a monolithic object precisely capturing the error pattern of an LLM is complex at best, often undecidable (Turing et al., 1936). Prefix filters offer a practical solution to this challenge, in that we do *not* require that filters be precise. Instead, what matters is that the filter is faithful to the *actual distribution* of the target LLM; i.e., it is OK that the filter excludes some outputs that are valid in principle, as long as these outputs are generated *very rarely* by the LLM in question.This observation allows prefix filters to be simple, efficient, and easy-to-learn, all of which are key desiderata.

PALLA: An algorithm for learning error patterns. We propose an algorithm PALLA for learning error patterns from concrete mistakes that LLMs make: by (*i*) unconditionally sampling the LLM on domain-specific tasks, (*ii*) identifying the bad outputs via an external oracle, (*iii*) grouping the errors, then (*iv*) querying a separate synthesizer on the bad outputs to distill error patterns as concrete objects. Figure 1 visualizes PALLA at a high level. We use an external LLM as the synthesizer in step 4.[1] For example, one might learn filters for TypeScript by using the TypeScript compiler as an oracle and asking an external LLM to characterize the samples rejected by the compiler.

Evaluating PALLA. We evaluate prefix filters and PALLA across 4 domains (generating (*i*) functions in MLIR, (*ii*) chemically valid molecules, (*iii*) natural language summaries, (*iv*) TypeScript functions) and 7 LLMs (Qwen2.5-1.5B/7B/14B (Yang et al., 2025b) and 3-8B (Yang et al., 2025a), Llama3.1-8B (Grattafiori et al., 2024), and Gemma3-12B/4-31B (Team et al., 2025; 2026)), with GPT-5.4-mini (OpenAI) as the synthesizer. The learned filters serve both goals of this paper. As a tool for analysis, they show that errors are clustered: e.g., on MLIR, the three most active filters for Qwen2.5-7B catch 51.7% of its errors. They also show that errors are largely model-specific: filters rarely transfer between models on MLIR and molecules, and they transfer well only on summaries, where every model leaks names in the same way. For constrained generation, filters learned from a modest number of samples raise validity: Qwen2.5-14B's MLIR compile rate rises from 27.5% to 71.0%, where grammar-constrained decoding lowers it to 19.5%, and Qwen2.5-1.5B's TypeScript compile rate rises from 16.0% to 70.0% with 132 learned filters, above the 45.4% of unconstrained Llama3.1-8B, a much larger model, although at roughly ten times the tokens per sample. A DPO baseline fine-tuned on the same samples also raises validity, but collapses output diversity: e.g., 31% of its MLIR outputs are distinct, against 92% with PALLA. The filters have a cost: together they reject about 5% of valid samples, and they help little on models that rarely make errors.

To summarize, our contributions are to (*i*) propose the idea of *prefix filters*, (*ii*) develop PALLA, an algorithm for learning prefix filters, and (*iii*) show that prefix filters learned by PALLA offer quantitative conclusions about LLM error patterns and are highly effective for constrained generation.

2 ERRORS, PROBLEM DEFINITION, AND RELATED WORK

We now formally define what it means for an LLM to make an 'error' on a certain domain, and state the *constrained generation problem*, which is to constrain the output of the LLM such that it does not make any errors. We use Σ to denote the tokens of an LLM, Σ^* to denote the set of strings obtainable by concatenating tokens, and $s \cdot s'$ for $s, s' \in \Sigma^*$ to denote the concatenation of s and s'.

[1] It may be possible to use deterministic program synthesizers (Park et al., 2025a; Kim et al., 2021) for this task, but we are unaware of any symbolic synthesizer targeting the task of mining patterns from a set of inputs.

Algorithm 1 Error-Driven Pattern Learning

1: **Input:** LLM $\mathcal{L}$, oracle $\mathcal{O}$, unconstrained sample budget B. **Output:** Set of error patterns Φ
2: $\Phi \leftarrow \emptyset$
3: $S \leftarrow \{x_i \sim \mathcal{L}\}_{i=1}^{B}$ ▷ Sample outputs from $\mathcal{L}$ to set S
4: $V \leftarrow \textsc{ValidSamples}(\mathcal{O}, S)$ ▷ Get valid samples from S
5: $I \leftarrow \textsc{InvalidSamplesByGroup}(\mathcal{O}, S)$ ▷ Group invalid samples from S according to cause
6: **for each** $I_e \in I$ ▷ For each invalid group I_e
7: $\quad \Phi \leftarrow \Phi \cup \textsc{LearnPatterns}(I_e, V)$ ▷ Learn patterns for group and add to filter set
8: **return** Φ

Definition 2.1 (Errors and Constrained Generation)**.** Let $\mathcal{L}$ be an LLM $\mathcal{L} : \Sigma^* \to \Delta(\Sigma^*)$ that takes as input a string p and outputs a distribution $\Pr_{\mathcal{L}}(\cdot \mid p)$ of strings. Let D be a *domain* and $\mathcal{O}$ be an *oracle* $\mathcal{O} : \Sigma^* \to \{\mathsf{true}, \mathsf{false}\}$ for D that classifies strings s into good samples ($\mathcal{O}(s) = \mathsf{true}$) and bad samples ($\mathcal{O}(s) = \mathsf{false}$). Let O_{ivld} be the set of bad samples $O_{ivld} = \{o_{ivld} \mid \mathcal{O}(o_{ivld}) = \mathsf{false}\}$.

Let P_D denote the set of prompts over the domain D. Given a prompt $p_D \in P_D$, if sampling from $\Pr_{\mathcal{L}}(\cdot \mid p_D)$ results in an invalid output $o_{ivld} \in O_{ivld}$, we say that $\mathcal{L}$ has made an *error*. The *constrained generation problem* for D is to restrict the image of $\mathcal{L}$ such that for any $p_D \in P_D$, $\Pr_{\mathcal{L}}(O_{ivld} \mid p_D) = 0$ (i.e., the probability of sampling any invalid string $o_{ivld} \in O_{ivld}$ is zero).

Rejection sampling, i.e., re-sampling from $\mathcal{L}$ until a good sample is drawn, is the most straightforward way to obtain an error-free set of outputs. However, rejection sampling is expensive, and is not always feasible online during test time, depending on the oracle. Existing work thus mainly attempts to solve the constrained generation problem in two ways: (*i*) constrained decoding, and (*ii*) fine-tuning.

Constrained decoding Constrained decoding refers to a family of techniques in which one starts with a grammar (Geng et al., 2023; Park et al., 2025b; Nagy et al., 2026) (or sometimes an automaton (Mündler et al., 2025)) that describes a set of acceptable strings. Using the automaton, one can then construct token masks that filter out tokens which are guaranteed not to have a valid continuation, guaranteeing that a generated sample is accepted.

Constrained decoding has many advantages: (*i*) modern implementations are highly efficient (Willard and Louf, 2023; AI, 2025), (*ii*) grammars can be reused across different LLMs, and (*iii*) it guarantees that samples are accepted by the grammar. However, grammars cannot in general precisely capture oracles due to decidability issues (Nagy et al., 2026), and constrained decoding thus must *overapproximate* oracles for properties such as type safety, resulting in invalid samples under constrained decoding. Constructing new grammars also remains a complex task that requires human expertise.

Fine-tuning Fine-tuning approaches to constrained generation directly update model weights so the output distribution better aligns with the target oracle. Reinforcement learning-style methods and related objectives (Ouyang et al., 2022; Stiennon et al., 2020; Rafailov et al., 2023b; Schulman et al., 2017; Christiano et al., 2017; Bai et al., 2022; Shao et al., 2024) have shown strong results in settings such as summarization and instruction following, and code-focused variants similarly use execution or test feedback to improve program generation (Le et al., 2022; Wei et al., 2025; Gehring et al., 2024). These methods are powerful, but typically require large amounts of data, compute, and careful design of the reward model.

PALLA and prefix filters alleviate the problems of existing approaches: PALLA does not require extra compute, data, or human intervention to learn prefix filters, and tailored per-domain-and-LLM error patterns creates a set of constraints that are much better aligned to the target LLM.

3 Error-Driven Pattern Learning and Prefix Filters

Algorithm 1 describes our approach for *error-driven pattern learning*, a process for learning the clustered errors from an LLM. At a high level, Algorithm 1 simply draws samples from $\mathcal{L}$ (line 3), divides them into valid and invalid samples (lines 4 and 5), then learns error patterns from the divided sets (line 7). Algorithm 1 itself does not force that the patterns are prefix filters; it represents a general algorithm for learning the error patterns of LLMs. The main idea in Algorithm 1 is that many oracles support *classifying* bad samples in depending on what the issue is; i.e., $\mathcal{O}$ returns a tuple

(v, e) where v indicates validity and e indicates the error if $v = \mathsf{false}$. Lines 6 and 7, group invalid samples into different sets I_e depending on their error e; line 8 learns filters separately for each group. The intuition is that information from the oracle can be exploited to further cluster invalid samples from the LLM $\mathcal{L}$; if an oracle does not have error support, Algorithm 1 can still proceed by placing all invalid samples in the same group.

3.1 Prefix Filters

We now propose prefix filters as concrete representations of the error patterns learned by LearnPatterns in Algorithm 1, that satisfies the need to (*i*) understand the error pattern of an LLM, and (*ii*) efficiently constrain the output of an LLM at test time.

Definition 3.1 (Prefix Filter)**.** A *prefix filter* is a binary classifier on strings: i.e., a function $\phi : \Sigma^* \to \{\mathsf{true}, \mathsf{false}\}$. For a given string s, we say that a filter ϕ *triggers* on s if $\phi(s) = \mathsf{false}$.

A filter ϕ is *sound* with respect to an oracle $\mathcal{O}$ iff for all s such that $\phi(s) = \mathsf{false}$, given any suffix string s', $\mathcal{O}(s \cdot s') = \mathsf{false}$—i.e., ϕ *only* identifies prefixes that are *guaranteed* to result in an invalid sample for any suffix s' according to the oracle $\mathcal{O}$.

A filter ϕ is *complete* with respect to an oracle $\mathcal{O}$ iff for all s such that for all s', $\mathcal{O}(s \cdot s') = \mathsf{false}$, $\phi(s) = \mathsf{false}$—i.e., ϕ recognizes *all* samples s that are guaranteed to result in an invalid completion.

Completeness of individual filters is *not* a desideratum in our approach: our goal is to learn *only* filters that are relevant to the error pattern of an LLM, not those recognizing the entire space of errors.

On the other hand, soundness *is* a desideratum for individual prefix filters: filters should not ban valid samples from being generated. However, enforcing soundness precisely given an arbitrary oracle is often an undecidable task and results in very complex filters. Instead, we exploit the fact that prefix filters are learned per-domain-and-LLM to define a notion of *distributional soundness*, a key design choice in Palla as discussed in Section 1.

Definition 3.2 (Distributional Soundness)**.** Let $\mathcal{L}$ be an LLM, D a domain, $\mathcal{O}$ an oracle for D, and $\Pr_D$ a distribution of prompts considered over D. Let $\Pr_{\mathcal{L},D}$ define the distribution of *outputs* o obtained by showing the LLM $\mathcal{L}$ prompts p sampled from $\Pr_D$: $\Pr_{\mathcal{L},D}(o) = \sum_p \Pr_D(p) \cdot \Pr_{\mathcal{L}}(o \mid p)$.

We say that a prefix filter ϕ is *distributionally sound within* ϵ with respect to an LLM $\mathcal{L}$, a domain D, and an oracle $\mathcal{O}$ iff for all samples s such that $\phi(s) = \mathsf{false}$ and suffixes s' such that $\mathcal{O}(s \cdot s') = \mathsf{true}$, $\sum_{s \cdot s'} \Pr_{\mathcal{L},D}(s \cdot s') < \epsilon$: i.e., ϕ mis-triggers on a valid generation of $\mathcal{L}$ with probability *at most* ϵ.

Example 3.3 (Distributionally sound). Palla learns the following filter for TypeScript & Qwen-1.5B:

```
def detect_unnecessary_whitespace(seq_prefix: str) -> bool:
    s = seq_prefix.strip()
    return True if (not s) else False
```

This filter bans whitespace-led samples and is unsound in general, but is distributionally sound on Qwen-1.5B: there, whitespace-led samples are *almost always* empty completions of a typed function body which fail to compile. The filter thus does not mis-reject valid samples, illustrating the main advantage of distributional soundness, accepting such interpretable filters tailored to each LLM.

In practice, we approximate distributional soundness by sampling a small set from $\Pr_{\mathcal{L},D}$.

3.2 Learning and Constrained Generation with Prefix Filters

Algorithm 2 illustrates how Palla learns candidate filters. QueryLargeLLM on line 4 first queries a large LLM on the given set of invalid and valid samples. We then Validate each individual filter ϕ_C for (*i*) distributional soundness, and (*ii*) whether the proposed filter captures invalid samples from I_e. Validated filters are added to the filter set Φ (line 10); if validation fails, we requery the large LLM with additional information E about why the filter ϕ_C failed validation (RequeryLargeLLM, line 9) up to a fixed number of times (line 6). Filters that fail validation after the retry limit are discarded.

Constrained generation using prefix filters After obtaining a set of prefix filters Φ via Algorithm 2, we use constrained adaptive rejection sampling (CARS) (Parys et al., 2025) to constrain the output of $\mathcal{L}$ when sampling. Algorithm 3 illustrates the decoding approach at a high level, where (*i*) we reject a

Algorithm 2 LEARNFILTERS: Querying an external LLM to learn prefix filters

Input: Invalid samples I_e, valid samples V, retry limit R. **Output:** Set Φ of prefix filters for I_e
$\Phi \leftarrow \emptyset$
$\Phi_C \leftarrow \text{QUERYLARGELLM}(I_e, V)$ ▷ Query large LLM to get candidate filter set Φ_C
for each $\phi_C \in \Phi_C$ ▷ For each filter $\phi_C \in \Phi_C$
for i **in range** R:
  $(v, E) \leftarrow \text{VALIDATE}(\phi_C, I_e, V)$ ▷ Validate ϕ_C
  if v **then break** ▷ If sound, break
  else $\phi_C \leftarrow \text{REQUERYLARGELLM}(\phi_C, I_e, V, E)$ ▷ Else requery LLM with error reason E
if $\text{VALIDATE}(\phi_C, I_e, V)$ **then** $\Phi \leftarrow \Phi \cup \{\phi_C\}$ ▷ If ϕ_C is sound, add ϕ_C to final set Φ
return Φ

Algorithm 3 CARS with Prefix Filters

Input: LLM $\mathcal{L}$, prompt p, prefix filters Φ. **Output:** Sample o
$T \leftarrow \emptyset$ ▷ T denotes token trie containing probabilities of token selection
while true:
  $o \leftarrow \epsilon, t \leftarrow \epsilon$ ▷ Set output o and next token t to ϵ
  while $t \neq$ EOF
    $t \leftarrow \text{SAMPLENEXTTOKEN}(\mathcal{L}, T, o); o \leftarrow o \cdot t$ ▷ Sample t from $\mathcal{L}$ using T
    if $\exists \phi \in \mathcal{F}$ such that $\phi(o \cdot t) = \text{false}$: ▷ If current prefix o is caught by a prefix filter
      $\text{UPDATETRIE}(T, o)$; **break** ▷ Update trie T to ban o and restart sample
  return o

sample being generated as soon as the prefix matches a filter in Φ, and (*ii*) when rejecting the sample, we update the probability trie (i.e., the logits) of the currently explored node to zero, and rebalance the probability trie with respect to this mass reduction. The latter rebalance allows subsequent sampling attempts to be more efficient; we refer the reader to (Parys et al., 2025) for details.

4 EXPERIMENTAL SETUP

We investigate how PALLA can learn error patterns as prefix filters in practice and how efficient the learned filters are for constrained generation across 7 local LLMs (Qwen2.5-1.5B, 7B, and 14B, Qwen3-8B, Llama3.1-8B, Gemma3-12B, and Gemma4-31B) and 4 domains:

MLIR: The task is to generate compile-valid torch-MLIR functions. Filters are learned from 50 samples, with the torch-MLIR compiler as the oracle. We evaluate compile rate on a set of 200 samples from Algorithm 3, against unconstrained and grammar-constrained baselines.

Molecules: The task is to generate chemically valid molecules (represented as SMILES strings (Weininger, 1988)) with high QED (drug-likeliness) scores (Bickerton et al., 2012). Filters are learned from 100 samples, using RDKit (RDKit) as the oracle for both validity and QED scores. We evaluate validity rate and average QED scores on a set of 200 samples with unconstrained and grammar-constrained decoding as baselines.

Dialogue summaries: The task is to summarize dialogues from the HR-MultiWOZ dataset (Xu et al., 2024) without leaking private data. Filters are learned from 500 samples (5 each for 100 dialogues), with a custom leak detector as the oracle. We evaluate the rate of summaries with no leaks on a set of 500 samples (10 each on a disjoint set of 50 dialogues) against an unconstrained baseline.

TypeScript: The task is to solve TypeScript benchmarks from MultiPL-E (Cassano et al., 2023). Filters are learned from 500 samples (5 each for 100 benchmarks), with the TypeScript compiler as the oracle. We evaluate compile rate on a set of 500 samples (10 each for a disjoint set of 50 benchmarks) with unconstrained generation as the baseline.[2]

We observe that for summary and TypeScript, the oracle captures domain validity rather than per-task correctness (e.g., the TypeScript compiler catches compile errors, not whether a compiling solution

[2]We do not use grammar-constrained decoding as a baseline for TypeScript because there is no publicly available grammar that fits our exact task (see Section C for details).

A. What filters does PALLA learn?

Domain	Model	What does the filter catch? [learned round, error group]	Triggers
MLIR	Qwen 7B	Undefined softmax-family op variants (e.g. `torch.aten.softmax`) [r1, unknown op]	130
MLIR	Qwen 14B	Wrong arity for `torch.aten.addmm` [r1, operand count]	70
Molecules	Llama 8B	Large overdecorated polyacyl scaffolds, a recurring low-QED molecular motif [r1, QED]	58
Molecules	Qwen 14B	Fused polycyclic aryl acetate/ester scaffolds with hydrophobic low-QED structure [r3, QED]	24
Summary	Qwen 1.5B	Language that explicitly leaks names (e.g., "their name is John") or attributes an action to a name (e.g., "approved by John") [r1, name leak]	179
Summary	Llama 8B	Patterns attributing a certain action to a person (e.g., "approved by John") [r1, name leak]	68
TypeScript	Qwen 14B	Reassignment of variables qualified by `const` [r1, TS2588]	645
TypeScript	Qwen 1.5B	`import` declarations inside solutions [r1, TS1003]	543

B. How well do the filters catch errors?

Domain		Qwen 2.5 1.5B	Qwen 2.5 7B	Qwen 2.5 14B	Llama 3.1 8B	Gemma 3 12B	Qwen 3 8B	Gemma 4 31B
MLIR (catch %)	Full	74.7	74.2	84.8	49.2	64.2	88.1	64.9
	Top-3	33.5	51.7	40.0	22.0	27.3	65.5	51.8
Molec. (catch %)	Full	44.9	58.1	46.5	59.6	74.3	45.5	56.4
	Top-3	16.8	35.5	38.4	48.2	62.4	38.2	43.6
Summ. (catch %)	Full	95.3	92.7	82.7	90.8	55.6	88.9	N/A
	Top-3	75.5	48.8	42.3	32.3	33.3	53.3	N/A
TS (catch %)	Full	99.3	71.3	74.4	94.1	4.0	55.4	82.2
	Top-3	75.2	16.5	35.9	57.5	0.0	30.4	79.5

C. How well do filters transfer?

Own	Cross	Ratio
71.5%	27.7%	0.39
55.0%	25.6%	0.47
84.3%	73.3%	0.87
68.7%	63.8%	0.93

N/A: no invalid Gemma 4 summaries.

Figure 2: Filters learned by PALLA and their relation to LLM error patterns. Panel A shows the most active filters per domain (Section 5.1). Panel B reports the percentage of invalid unconstrained samples caught by the full filter set (full) or the top-three active filters (top-3), (Section 5.2). Panel C reports mean capture rates on each model's own invalid samples (Own) and other models' invalid samples (Cross); Ratio is Cross/Own (Section 5.3). Section H reports confidence intervals.

solves the benchmark). This is an intentional choice to learn filters for the entire domain, although in principle, we could learn filters specialized to a task as well. In Section C, we also consider a fifth domain (SQL), which we do not include in the main text due to overlap with other domains.

Prefix filters are learned via QUERYLARGELLM in Algorithm 2 to synthesize filters of type STR $\rightarrow$ BOOL in Python. We use GPT5.4-mini as the large LLM; Section F shows an additional experiment using GPT-OSS-120B (OpenAI, 2025). For each domain-LLM pair, we performed *three rounds* of filter learning: first on unconstrained samples, then on the distributions induced by previously learned filters, because filters can shift the error patterns of models and expose different error modes. Details are provided in Section C.1.

5 RESULTS

Figure 2 presents a summarized analysis of the filters we learn, while Figure 3 presents results on constrained generation. We center our evaluation around 4 main questions. A detailed per-domain analysis of results can be found in Section E. Comparisons made in this section are statistically supported at 95% bootstrap confidence intervals following the method in (Berg-Kirkpatrick et al., 2012), unless otherwise noted. In this section, we say that a filter has caught an error if it triggers on line 7 of Algorithm 3, meaning that the filter has prevented an invalid sample from being generated.

5.1 LEARNABILITY: HOW WELL CAN PALLA DISTILL ERROR PATTERNS AS PREFIX FILTERS?

Panel A in Figure 2 lists the most active filters for each domain: e.g., the Qwen-7B MLIR filter caught 130 invalid samples while generating the evaluation set of 200 MLIR samples via Algorithm 3.

Most active filters were learned in round 1, but shifts in the error distribution also led to learning in rounds 2 and 3. For example, Qwen-1.5B's round-2 TypeScript filter targets TS2304 (invalid names). After the first round, TS1005 (syntax) errors fell from 260 to 7 of 500 samples, while TS2304 errors rose from 23 to 51, providing more examples from which PALLA could learn a filter.

Panel B illustrates that the learned filters are effective at catching invalid samples from the unconstrained distribution of the LLM: for example, filters learned for MLIR & Qwen-7B were able to detect 74.2% of errors generated by the unconstrained model. The high capture rates in Panel B confirm that PALLA is capable of learning error-specific prefix filters; the only exceptions are TypeScript & Gemma3-12B and summaries $ Gemma4-31B, where prefix filters failed because Gemma almost always generated valid samples (95.0 % compile rate on TypeScript, 100.0 % safety on summaries) and there were too few invalid examples to learn filters from.

We approximated distributional soundness by replaying LLM-proposed filters on valid learning-phase samples and external reference solutions, rejecting filters with a higher than 1% false positive rate. We observe that even if each individual filter rejects less than 1% of valid samples, their union may yield a higher false positive rate; in our experiments, the entire set of learned filters rejected on average 5% of valid samples. PALLA thus does lose some valid samples due to the prefix filters, but we argue this is an acceptable tradeoff given the benefits of learned filters during generation (Section 5.4).

Conclusion: PALLA is capable of learning prefix filters that capture a wide portion of the per-domain-and-LLM error pattern.

5.2 CLUSTERING: DO LLMS ACTUALLY MAKE A SMALL SET OF CLUSTERED ERRORS?

Panel B validates our cluster hypothesis: the top 3 filters for each domain-and-LLM catch a significant portion of errors. For example, Qwen-7B learned 20 MLIR filters, but its top three caught *51.7%* of errors, or 70% of the errors caught by the full filter set. An example filter for MLIR & Qwen-7B is:

```
def filter_unknown_torch_aten_softmax_family_op(text: str) -> bool:
    import re
    return re.search(
        r'(?m)^\s*%[\w.]+\s*=\s*torch\.aten\.(?:softmax|log_softmax|logsumexp)\s',
        text) is not None
```

This filter just rules out three concrete undefined MLIR operators (`torch.aten.softmax`, `log_softmax`, or `logsumexp`), but triggers on 35.0% of the errors from Qwen-7B. Another example for Molecules & Qwen-14B is:

```
def filter_polycarbonyl_rich_scaffold(smiles_prefix: str) -> bool:
    return smiles_prefix.count("C(=O)") >= 3
```

This filter simply rejects molecules that already have 3 carbonyl (`C(=O)`) instances but triggers on 25.0% of invalid attempts. That these simple filters trigger on over 20% of errors indicates that LLMs indeed have clustered error modes that can be detected and prevented with prefix filters.

Conclusion: LLMs do make clustered errors that can be caught with just a few filters. There do remain errors outside of the clustered modes; prefix filters alone cannot capture the entire set.

5.3 FILTER TRANSFERABILITY: CAN FILTERS LEARNED FOR ONE MODEL BE APPLIED TO ANOTHER? OR, DO DIFFERENT LLMS MAKE SIMILAR MISTAKES?

Panel C investigates how well filters learned from one model *transfer* to another model. MLIR and molecules display a very low ratio of transferability, illustrating that the error patterns of LLMs differ in these domains. Transferability seemed to have little correlation with model lineage (detailed cross-filter capture rates are availble in Section E), suggesting that errors were model-specific instead of lineage-specific. On the other hand, the summary domain shows a high level of transferability: this is because across all models, name leaks are by far the most common mode of error. For example,

```
def filter_name_named_or_identified_as(seq_prefix: str) -> bool:
    return bool(re.search(
        r'\b(?:named|called|identified as|referred to as)\s+'
        r'[A-Z][a-z]+(?:\s+[A-Z][a-z]+){0,2}\b', seq_prefix))
```

This filter was learned for Qwen-1.5B, but applies broadly across all models as it identifies common name leak patterns: a lead-in (e.g., `named`), followed by the name, which first character is capitalized.

For TypeScript the results also suggest transferability, but they are partly inflated by filters that are distributionally sound for one model but unsound for others. For example, the `whitespace` filter

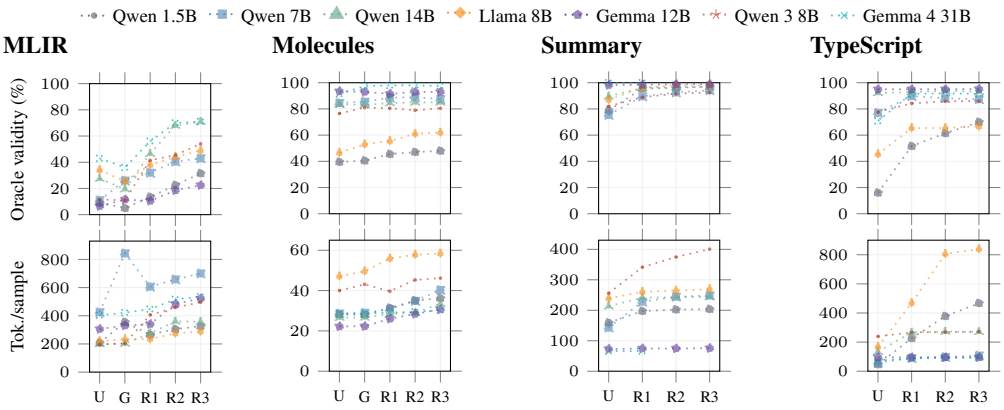


Figure 3: Validity and token costs per sample across domains and LLMs. Top row plots validity rates according to the oracle for each domain; bottom row plots average tokens required to generate a sample. X-axis ticks are ordered (U)nconstrained, (G)rammar-constrained (for MLIR and Molecules), and (R)ounds 1, 2, and 3 of filters. An upward right trend indicates that validity (top row) / tokens per sample (bottom row) is increasing as filters are added.

for Qwen-1.5B discussed in Example 3.3 is *unsound* for Gemma, which often inserts newlines before valid solutions; this filter thus fires on most Gemma samples and inflates the cross-validity ratio.

Conclusion: Filter transferability depends on the domain. In some domains, LLMs are likely to make similar mistakes; in others, their error patterns are significantly different.

5.4 Efficacy: How effective are learned filters for constrained generation?

Validity gains. Figure 3 summarizes how effective learned filters are for constrained generation. In most cases, PALLA shows a clear increase in oracle validity compared to unconstrained, or grammar-constrained decoding (GCD): e.g., TypeScript validity for Qwen-1.5B jumps from 16.0% to 70.0% after three rounds of filter learning. The graphs also show that for most cases, each individual round of filter learning helps increase the validity rate (e.g., for MLIR). In particular, GCD suffers for MLIR because grammars cannot prevent prevalent error modes such as using undefined operators. Prefix filters prevent such error modes and achieve a much higher compile rate. We observe that prefix filters can also be used in *tandem* with GCD, to prevent errors that GCD alone cannot capture; a small extension experiment is provided in Section G.

The exception to validity increase is when the model is good enough that there are too few errors to learn invalid filters (e.g., Gemma-12B and TypeScript).[3] For Typescript & Llama-8B, the first round of filters resulted in a significant improvement, but subsequent rounds failed to increase validity rate.

We note that prefix filters do not increase validity rates by sacrificing other quality metrics. For example, in the Summary domain, the average number of non-protected data fields each summary contained remained stable (i.e., the filters precisely targeted only prefixes that led to actual leaks). For TypeScript, sample-level solve rates (the percentage of samples that yielded a successful solution) increased for models where validity increased (Qwen-1.5B and 7B, Llama-8B) while staying stable for other models. This analysis shows that the prefix filters are not simply collapsing the samples towards functions that compile trivially, and that the prefix filters also help increase downstream task performance in cases where oracle validity does not immediately imply final task correctness. Detailed per-domain results including secondary quality metrics can be found in Section E.

Baseline Comparison. Across all four domains, we trained a QLoRA-DPO (Dettmers et al., 2023; Rafailov et al., 2023a) baseline[4] on same-prompt pairs of oracle-valid and oracle-invalid round-0 generations, using the same samples as PALLA. Detailed configurations are in Section C.2.

[3]Validity increase for Molecules & Qwen-1.5B is statistically insignificant at the 95% interval, but becomes significant at 90%. All other experiments saw an increase when the unconstrained model had $< 80\%$ validity.

[4]On Qwen-2.5 1.5B/7B/14B, Llama3-8B, and Gemma3-12B as a representative model set.

Figure 4 compares the results of unconstrained, PALLA-constrained, and LoRA generation. LoRA achieved high validity but collapsed output diversity: for MLIR, only 31% of outputs were distinct on average, compared with 92% for PALLA. LoRA seems to maintain a high ratio of distinct outputs for summaries, but the average percentage of non-protected data fields collapsed by 20.6 pp (compared to only 0.4 pp for PALLA): LoRA filtered over-aggressively, often generating useless samples. These results suggest that with sparse valid training samples, DPO can concentrate on a few oracle-valid outputs and sacrifice coverage. PALLA instead selectively removes failure patterns and prevents collapse.

Domain	Oracle validity (%) U / P / L	Distinct outputs (%) U / P / L
MLIR	20.4 / 48.6 / 89.0	98.0 / 92.0 / 30.9
Molecules	69.5 / 75.1 / 69.1	91.4 / 90.0 / 36.9
Summary	85.8 / 97.0 / 99.9	100.0 / 100.0 / 100.0
TypeScript	57.7 / 79.3 / 85.9	61.6 / 75.4 / 62.5

Figure 4: Comparison of validity rates and distinct outputs for unconstrained (U), PALLA-constrained (P), and QLoRA-DPO (L) generated samples. Values are averages across evaluated models.

Conclusion: For cases where the base model has room for improvement in terms of validity rate, prefix filters are highly effective at constraining the output of the LLM towards valid outputs. PALLA does *not* require many invalid samples to learn filters, making it especially attractive in low-data settings where fine-tuning data is hard to obtain.

Overhead. Rejection sampling increases tokens per sample (Algorithm 3), but overhead is modest relative to validity gains in most domains: On average, MLIR had an increase of 60.5% in tokens per sample, molecules had an increase of 31.3%, and the summaries domain had an increase of 26.5%. TypeScript had the largest overhead: 914.1% for Qwen-1.5B and 400.2% for Llama-8B. Qwen-1.5B made many errors, requiring frequent rejections. Llama-8B often errored near the end of a sample, e.g., by adding explanatory text after the solution, making rejections expensive for Algorithm 3.

Across three rounds, GPT5.4-mini filter synthesis consumed, on average, 3.0M/0.9M input/output tokens for MLIR, 0.6M/1.1M for molecules, 4.5M/1.8M for summaries, and 3.2M/3.1M for TypeScript. Filter generation cost roughly $5 for MLIR to $15 for TypeScript.

Conclusion: Tokens per sample overhead remains modest in general, but can become higher in a few cases that prefix filters are not well-suited for. Filter generation overhead is also manageable.

6 LIMITATIONS AND DISCUSSION

Limitations. PALLA is ineffective on already competent models that provide few invalid examples for learning prefix filters. Larger or task-specific learning sets for filters may alleviate this problem, but both approaches would also increase the learning cost for PALLA. Another limitation is that prefix filters may be unlearnable if the error is not soundly prefix-expressible: e.g., if `list.print` is not a valid operator but `list.printAll` is, then PALLA may be unable to learn prefix filters that prevent `list.print` from occurring. An example is illustrated in Section E.

Other Related Work on Constrained Generation. Verbalized RL (Shinn et al., 2023) translates outputs from oracles into text that is prompted to the LLM on subsequent queries, attempting to prevent the LLM from making similar mistakes via prompting. Some variants of grammar-constrained decoding (Geng et al., 2023) have elements that attempt to learn the grammar automatically, similar to how we learn prefix filters. In contrast, our main focus is on analyzing the error patterns of LLMs *themselves*; constrained generation is an example of how this information can be exploited.

Future work. Prefix filters open several directions for future work: e.g., generating error-focused datasets for fine-tuning, to remove recurring failure modes with less data and compute. Another extension is towards agentic systems, where often-repeated mistakes such as malformed tool calls could be learned as prefix filters and prevented.

7 CONCLUSION

We proposed *prefix filters*, symbolic functions that capture per-domain-and-LLM error patterns, and PALLA, a system for learning prefix filters. We showed that prefix filters can be learned and used to constrain LLM efficiently in practice; prefix filters provided insight into the error patterns of different LLMs, showing that different LLMs make different mistakes in many cases. We anticipate that prefix filters have many future applications such as fine-tuning or making agentic behavior reliable.

AI USE STATEMENT

In this work, we used generative AI tools to help design, implement and analyze the experiments in Section 5, investigate related work in Section 6, and edit figures and writing to improve readability. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This paper presents work whose goal is to advance the field of Machine Learning. We do not believe that our work raises questions related to the Code of Ethics.

REPRODUCIBILITY STATEMENT

We submit as supplementary material our source code and the raw collected from the llms while running the experiments in the main text.

REFERENCES

Smiles grammar. URL https://github.com/genlm/genlm-eval/blob/main/assets/molecular_synthesis/smiles.lark.

tree-sitter-typescript. https://github.com/tree-sitter/tree-sitter-typescript. Accessed 2026-05-06.

Typescript error codes. URL https://github.com/microsoft/TypeScript/blob/main/src/compiler/diagnosticMessages.json.

Lark - a parsing toolkit for python. https://github.com/lark-parser/lark, 2025. Accessed 2026-05-06.

Guidance AI. llguidance: Super-fast structured outputs, 2025. URL https://github.com/guidance-ai/llguidance. Accessed: 2025-09-23.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

Taylor Berg-Kirkpatrick, David Burkett, and Dan Klein. An empirical investigation of statistical significance in nlp. In *Proceedings of the 2012 joint conference on empirical methods in natural language processing and computational natural language learning*, pages 995–1005, 2012.

G. Richard Bickerton, Gaia V. Paolini, Jérémy Besnard, Sorel Muresan, and Andrew L. Hopkins. Quantifying the chemical beauty of drugs. *Nature Chemistry*, 4(2):90–98, 2012. doi: 10.1038/nchem.1243.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023.

Paul F. Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *arXiv preprint arXiv:1706.03741*, 2017.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115, 2023.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*, 2024.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured nlp tasks without finetuning. *arXiv preprint arXiv:2305.13971*, 2023.

GenLM Consortium. GenLM: Language model probabilistic programming. https://genlm.org/, 2026. Accessed 2026-05-06.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, and Abhishek Kadianand others. The Llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. Semantics-guided synthesis. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–32, 2021.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. doi: 10.1109/CGO51591.2021.9370308.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357, 2023.

Niels Mündler, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. Type-constrained code generation with language models. *Proceedings of the ACM on Programming Languages*, 9(PLDI):601–626, 2025.

Shaan Nagy, Timothy Zhou, Nadia Polikarpova, and Loris D'Antoni. Chopchop: A programmable framework for semantically constraining the output of language models. *Proc. ACM Program. Lang.*, 10(POPL), January 2026. doi: 10.1145/3776708. URL https://doi.org/10.1145/3776708.

OpenAI. Introducing gpt-5.4 mini and nano. URL https://openai.com/index/introducing-gpt-5-4-mini-and-nano/.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Kanghee Park, Xuanyu Peng, and Loris D'Antoni. Loud: Synthesizing strongest and weakest specifications. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA1):956–983, 2025a.

Kanghee Park, Timothy Zhou, and Loris D'Antoni. Flexible and efficient grammar-constrained decoding. In *Forty-second International Conference on Machine Learning*, 2025b. URL https://openreview.net/forum?id=L6CYAzpO1k.

Paweł Parys, Sairam Vaidya, Taylor Berg-Kirkpatrick, and Loris D'Antoni. Constrained adaptive rejection sampling, 2025. URL https://arxiv.org/abs/2510.01902.

Daniil Polykovskiy, Alexander Zhebrak, Benjamin Sanchez-Lengeling, Sergey Golovanov, Oktai Tatanov, Stanislav Belyaev, Rauf Kurbanov, Aleksey Artamonov, Vladimir Aladinskiy, Mark Veselov, Artur Kadurin, Simon Johansson, Hongming Chen, Sergey Nikolenko, Al'an Aspuru-Guzik, and Alex Zhavoronkov. Molecular sets (MOSES): A benchmarking platform for molecular generation models. *Frontiers in Pharmacology*, 11:565644, 2020. doi: 10.3389/fphar.2020.565644.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023a.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023b.

RDKit. RDKit: Open-source cheminformatics. `http://www.rdkit.org`. Accessed: 2025-09-20.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in neural information processing systems*, 33:3008–3021, 2020.

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alex Feng, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucińska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Noveen Sachdeva, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Kat Black, Nabila Babar, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley, Slav Petrov, Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry, Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin,

Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. Gemma 3 technical report, 2025. URL https://arxiv.org/abs/2503.19786.

Gemma Team, Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, Ian Ballantyne, Cormac Brick, Victor Cărbune, Michelle Casbon, et al. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026.

Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

Sairam Vaidya, Marcel Böhme, and Loris D'Antoni. Bootstrapping fuzzers for compilers of low-resource language dialects using language models. *arXiv preprint arXiv:2512.05887*, 2025.

Anjiang Wei, Tarun Suresh, Huanmi Tan, Yinglun Xu, Gagandeep Singh, Ke Wang, and Alex Aiken. Supercoder: Assembly program superoptimization with large language models. *arXiv preprint arXiv:2505.11480*, 2025.

David Weininger. SMILES, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of Chemical Information and Computer Sciences*, 28(1):31–36, 1988. doi: 10.1021/ci00057a005.

Brandon T Willard and Rémi Louf. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*, 2023.

Weijie Xu, Zicheng Huang, Wenxiang Hu, Xi Fang, Rajesh Cherukuri, Naumaan Nayyar, Lorenzo Malandri, and Srinivasan Sengamedu. HR-MultiWOZ: A task oriented dialogue (TOD) dataset for HR LLM agent. In *Proceedings of the First Workshop on Natural Language Processing for Human Resources (NLP4HR 2024)*, pages 59–72, St. Julian's, Malta, March 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.nlp4hr-1.5. URL https://aclanthology.org/2024.nlp4hr-1.5/.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025b. URL https://arxiv.org/abs/2412.15115.

A Model Checkpoints

We used the following models in our evaluation:

Model	URL	Commit
Qwen 1.5B	`https://huggingface.co/Qwen/Qwen2.5-1.5B-Instruct`	`989aa7`
Qwen 7B	`https://huggingface.co/Qwen/Qwen2.5-7B-Instruct`	`a09a35`
Qwen 14B	`https://huggingface.co/Qwen/Qwen2.5-14B-Instruct`	`cf98f3`
Llama 8B	`https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct`	`0e9e39`
Gemma 12B	`https://huggingface.co/google/gemma-3-12b-it`	`96b6f1`

For the large LLM in QueryLargeLLM, we queried OpenAI gpt-5.4-mini, accessed April 21–30, 2026.

B Hardware and Software

Experiments were conducted on a Ubuntu 20.04 LTS server equipped with a AMD EPYC 7282 CPU (32 physical cores total at up to 2.8GHz), 512GB of RAM, and NVIDIA LS40 GPUs. Each experiment was run on a single GPU. We note that Palla or constrained sampling using prefix filters does not require additional hardware outside of a standard CPU; the GPU was used only for local model inference. Our implementation is based on Python 3.12.12, PyTorch 2.10.0 with CUDA 12.8, HuggingFace Transformers 5.1.0, RDKit 2025.9.5, and torch-mlir 20260314.751.

C Details on the experimental setup

We provide more details on our experimental setup. We enforce a 1% error rate for distributional soundness, both on the model's own distribution, and also on a external dataset of validated solutions for each task. We find that augmenting the model's own distribution with an external solution set helps block filters that are distributionally sound on the model's distribution, but would trigger on candidate solutions that arise from shifted distribution obtained post-filtering.

C.1 Detail on Each Domain

MLIR For MLIR, the local LLM is tasked with generating a valid (compilable) MLIR function without additional constraints. Our oracle for this task is the MLIR compiler `torch-mlir-opt`, which returns true if the sample successfully compiles and false (along with an error message) if the sample fails to compile. We sample 50 MLIR programs for the learning filters and 200 for evaluating constrained generation on the learned filters. Learned filters are validated on valid learning samples generated by the local LLM in addition to a fixed set of valid MLIR programs. Our baselines for this domain are unconstrained generation, as well as grammar-constrained decoding using a Lark grammar we adapted from one previously used in other projects on constrained decoding (Vaidya et al., 2025; Parys et al., 2025).

Error Classes. `torch-mlir-opt`, like many other compilers, emits an error message detailing why the program cannot compile. If a learning sample is invalid, we take the *first* error message emitted by `torch-mlir-opt` and use it to classify sample into one of 8 error classes (use of undefined custom operators, operator arity errors, type errors, SSA errors, syntax errors, invalid type annotations, invalid literals, and other miscellaneous errors).

Molecules For molecules, the local LLM is tasked with generating a chemically valid molecule in the SMILES string format without additional constraints. We sample 100 molecules for the learning phase and sample 200 molecules after learning prefix filters as the testing phase.

We use *two* oracles for the molecules task: (*i*) RDKit (RDKit), which returns a binary true / false indicating whether the molecule is chemically valid, and (*ii*) QED score (Bickerton et al., 2012), which is a real value between 0 and 1 indicating how good the sample is as a drug. QED score is converted as a binary oracle for filter learning by analyzing the distribution of QED scores generated

by the model, and setting those with a higher percentile as good samples, while setting those with a lower percentile as bad samples. Learned filters are validated against valid (chemically valid / high QED-scoring) learning samples and an external set of 1000 molecules from the MOSES SMILES dataset (Polykovskiy et al., 2020). Our baselines for this domain are unconstrained generation, as well as grammar-constrained decoding using a SMILES grammar adapted from GenLM (smi; GenLM Consortium, 2026).

We set both QED score and validity as filter targets to demonstrate that PALLA can learn prefix filters that optimize over multiple targets at once, not just a single target. We note that the authors are unfamiliar with molecular chemistry: the molecules domain also serves as an example of how PALLA can be used without any domain expertise to enhance the performance of LLMs.

Error Classes. Similar to a program compiler, RDKit provides justification on why a given sample is chemically invalid (e.g., a chemical ring is never closed); we use this information to classify chemically invalid samples into 8 categories. For QED score, all 'bad' samples (low-QED scoring molecules) share a single error group.

Summaries In the summaries domain, the local LLM is given as input a dialogue between an HR department and a worker and asked to summarize the dialogue without leaking any private information. The HR dialogues are took from the HR-MultiWOZ dataset (Xu et al., 2024), consisting of 550 HR dialogues, each of which are annotated with information fields that the dialogue contains (e.g., name, email, phone). We define private information for this task as the fields `name`, `phone`, `id`, `email`, and `contact_bundle` (which contains miscellaneous contact info); other fields exist in the dataset as well (e.g., `time_off`, `training_request`, etc.). The goal of the local LLM is to generate a summary of the dialogue that does *not* contain any of the 5 protected fields but contains as many of the other fields as possible. The oracle takes the values of the protected fields within the dataset and returns `false` if any of the concrete values are present within the generated summary.

We take 100 dialogues from the dataset and generate 5 summary samples per dialogue for learning, then take a disjoint set of 50 dialogues and generate 10 samples each for testing constrained generation (thus keeping the learning and testing sets separate). Learned filters are validated against the non-leaking summaries generated by the local model as well as a set of 388 safe reference summaries generated by GPT5.5 and validated externally. We use only unconstrained generation as a baseline, as it is hard to develop a formal grammar for natural language summaries; the summaries domain illustrates that prefix filters can be learned and applied to constrained generation even in such fuzzy domains.

Error Classes. Invalid samples in the HR domain (those that leak sensitive information) are classified according to which field of sensitive information they leak for a total of 5 groups. Note that a single sample can leak multiple fields at once, in which case the offending sample will be placed in *all* relevant buckets, not just one.

TypeScript For TypeScript, we take the MultiPL-E benchmark (Cassano et al., 2023), which contains 390 benchmark entries, each with the goal of synthesizing a TypeScript function that meets a natural-language specification. The task of the local LLM is to solve these benchmarks. Our oracle for this domain is the TypeScript compiler, which returns `true` if the generated sample successfully compiles and `false` otherwise. We do *not* take whether the local LLM succeeded in generating a *correct* solution for a given problem as our validity signal—doing so would learn prefix filters that are specialized for solving a particular TypeScript problem, instead of generating valid TypeScript in general.

We take 100 problems from the MultiPL-E benchmark and generate 5 samples each for learning prefix filters, then take a disjoint set of 50 problems and generate 10 samples each for testing constrained generation. Learned filters are validated against the compiling learning samples generated by the local LLM as well as an external set of 390 solutions for the MultiPL-E dataset. We take only unconstrained generation as the baseline for TypeScript.

We do not compare against grammar-constrained decoding, despite the fact that TypeScript has been studied previously in grammar-constrained settings, for two main reasons. First, as far as the authors are aware, there is no official TypeScript grammar; the official specification of TypeScript is given via the implementation of the TypeScript compiler instead. Second, while there do exist

third-party grammars (e.g., (tre)) for TypeScript, these grammars are not directly aligned with our benchmark tasks of completing a TypeScript function whose signature has already been defined; our implementation also depends specifically on Lark (lar, 2025), and also includes prompting with triple-quote-fencing (‴ ‴) to increase the performance of the local LLMs. While it would be possible in theory to adapt existing implementations towards our particular scenario, doing so would require nontrivial effort. PALLA and prefix filters propose a path towards constrained generation *without* needing such effort, which is why we do not compare against grammar-constrained decoding for TypeScript.

Error Classes. Invalid TypeScript samples are categorized by the first error code that the TypeScript compiler emits (tse). There are many TypeScript error codes which may lead to a very large set of error classes, but in practice we find that invalid samples from the LLM exercise just a few (e.g., TS1005: token expected or TS2339: property does not exist on type), keeping the set of error classes small and relatively concentrated.

SQL We introduce SQL queries as an additional domain, which is not presented in the main text due to overlaps with the other domains. For the SQL domain, the goal is to synthesize valid SQL queries that solve a certain natural-language SQL task, such as retrieving a certain value. Our dataset is the `formula_1` database from the BIRD-Mini Dev PostgreSQL benchmark (Li et al., 2023), a 13-table database with 94 columns, 19 foreign-key links, and roughly 493000 rows, paired with 66 query tasks over the `formula_1` database also from BIRD. Our validity oracle is the PostgreSQL execution engine.

We use 26 tasks for learning filters and generate 5 samples per learning task for a total of 130 learning samples, and the remaining 40 tasks with 10 samples per task for a total of 400 testing samples. Like TypeScript, we learn filters targeting the validity of a query itself, not whether the query solves a specific task, in order to learn filters that are broadly applicable to the entire domain. Candidate filters are validated against the model's own learning set plus an external solution set of 400 valid SQL queries. Our baselines for the SQL domain are unconstrained generation and grammar-constrained decoding.

Error Classes. The PostgreSQL engine also returns a variety of errors for invalid queries like the TypeScript compiler. For the SQL domain, we group the errors into seven groups: parse errors, scope errors, aggregation / grouping errors, function signature errors, type errors, queries that attempt to access a missing relation or column, and other miscellaneous errors.

C.2 QLoRA-DPO Baseline

For the QLoRA-DPO baseline compared in Section 5.4, we trained a rank-16 LoRA on all attention and MLP projections, with $\alpha = 32$, DPO $\beta = 0.1$, learning rate $= 5 \cdot 10^{-5}$, and three epochs. LoRAs were trained on same-prompt preference pairs from the round-0 learning phase, exactly the same set that PALLA learned its filters from. We did not perform separate hyperparameter tuning. We trained LoRAs on Qwen-2.5 1.5B/7B/14B, Llama3-8B, and Gemma3-12B as a representative set, Figure 4 reports an average over these 5 models. LoRA fine-tuning on these models required 43.8 GPU hours in total on NVIDIA LS40 GPUs.

D Prompts

Prompts to query the small / large LLMs are included as supplementary material with our submission.

E Extra per-domain experimental results

We provide a detailed per-domain analysis of our experimental results in Section 5.

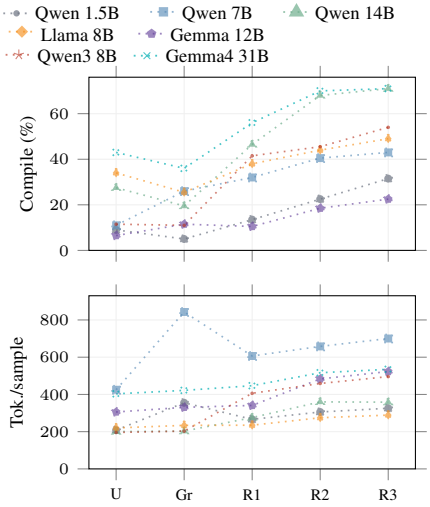
E.1 MLIR

Figure 5 summarizes the results for the MLIR domain, with Panel A on the left describing the most active filters per model, and Panel B comparing compile rates and tokens / sample for each mode

A. Most-triggered MLIR prefix filters per-model

Model	Top triggered filters [learned round]
Qwen 1.5B	Malformed `func.func` headers [r0] (63 events, 18.2%)
	Invented custom `torch.*custom*` ops [r0] (52 events, 15.0%)
Qwen 7B	Unsupported softmax-family op forms [r0] (130 events, 33.2%)
	Unsupported miscellaneous ATen op names [r0] (57 events, 14.5%)
Qwen 14B	Wrong arity for `torch.aten.addmm` [r0] (70 events, 23.2%)
	Wrong arity for other fixed-arity ATen ops [r1] (33 events, 10.9%)
Llama 8B	Additional unsupported ATen op names [r1] (14 events, 13.3%)
	Invented or unsupported Torch op names [r0] (12 events, 11.4%)
Gemma 12B	Unsupported exact custom op names [r0] (48 events, 14.4%)
	Too many operands for fixed-arity ops [r0] (47 events, 14.1%)
Qwen 3 8B	Undefined `torch.aten.softmax` op [r0] (116 events, 26.8%)
	Wrong operand counts for known ATen ops [r0] (114 events, 26.3%)
Gemma 4 31B	Tensor operands passed to scalar-form `add` [r1] (39 events, 37.5%)
	Wrong operand counts for fixed-arity ops [r0] (32 events, 30.8%)

B. MLIR performance by method



C. Filter capture rates across models

	Q2.5 1.5B	Q2.5 7B	Q2.5 14B	L3.1 8B	G3 12B	Q3 8B	G4 31B
Q2.5 1.5B	74.7	19.7	6.2	14.4	15.0	40.1	28.9
Q2.5 7B	23.1	74.2	34.5	16.7	24.6	42.9	38.6
Q2.5 14B	23.1	11.8	84.8	19.7	19.3	22.0	29.8
L3.1 8B	35.7	19.1	45.5	49.2	26.7	46.9	36.0
G3 12B	36.8	44.9	36.6	12.9	63.6	39.5	55.3
Q3 8B	33.5	39.9	31.0	9.8	20.3	88.1	39.5
G4 31B	11.5	41.0	14.5	3.8	16.0	35.6	64.9

Figure 5: Summarized results for prefix filters over the MLIR domain. Panel A shows the descriptions of the two most active filters for each model; parentheses report the number of triggers (i.e., the number of errors caught by the filters) and the percentage of triggers that each filter contributed to the entire model (e.g., the first filter for Qwen-1.5B was responsible for catching 20.8% of errors made by the LLM). Panel B plots compile rate and consumed tokens per sample. Panel C measures filter transferability: each entry reports the percentage of bad unconstrained samples from the column model that are caught by at least one of the filters learned from the row model. Diagonal entries show the rate of bad samples matched by the model's own learned filters; off-diagonal entries show cross-model transfer. Darker red indicates weaker transfer relative to the row model's own coverage.

of generation. The high trigger rates for the top filters support our hypothesis that errors made by LLMs are mostly clustered: e.g., for Qwen-7B, the following top filter was responsible for 33.2% of all filter trigger events:

```
def filter_unknown_torch_aten_softmax_family_op(text: str) -> bool:
    import re
    return re.search(
        r'(?m)^\s*%[\w.]+\s*=\s*torch\.aten\.(?:softmax|log_softmax|logsumexp)\s',
        text,
    ) is not None
```

This filter is very simple, just ruling out three kinds of MLIR prefixes that contain `torch.aten.softmax`, `log_softmax`, or `logsumexp`, all three of which are unsupported (undefined) MLIR operators. However, Qwen-7B generates programs containing these operators exceedingly often, as shown by the high trigger rate of this filter. Other models show similar clustered behavior around a few error modes: across all seven models, the top 3 and top 5 named filters

accounted for an average of 53.3% and 69.2% of final-round testing trigger events, respectively. Models learned 23.4 filters on average, ranging from 12 for Gemma4-31B to 32 for Qwen2.5-14B.

In MLIR, the learned filters lead to a significant increase in the compile rate of generated samples, with the most significant increase observed in Qwen-14B: from 27.5% unconstrained to 71.0% after three rounds of filter learning. Grammar-constrained decoding performs poorly for MLIR, as grammar-constrained decoding fails to prevent the most prominent failure modes of the model (e.g., using undefined `torch.aten` operators). The increased compile rate does come at a cost of requiring more tokens per sample to generate due to CARS, plotted at the bottom of Panel B, with a mean per-model increase of 69.7% from unconstrained to final-round generation across all seven models.

Panel C in Figure 5 investigates *filter transferability* across different models, where the filters of a row are applied against the unconstrained samples of the column models (e.g., 74.7% of Qwen-1.5B's bad unconstrained samples are caught by its own prefix filters). The table shows that filter transferability is generally poor. In other words, *error modes differ* quite severely between models, even between models from the same family but of different size—for example, filters from Qwen-14B filters match 84.8% of its own bad samples, but cover only 23.1% and 11.8% of bad samples from Qwen-1.5B and 7B. The outlier is filters from Llama-8B catching bad samples from Qwen-14B relatively well—this is because Llama also contains a filter that catches arity issues in `torch.aten` operators, the most dominant error mode for Qwen-14B.

E.2 Molecules

Figure 6 summarizes the results for the molecules domain, where we learn filters for both chemical validity and QED score. Results for molecules display a similar trend to those for MLIR, except that some models (Qwen 14B in particular) are already highly competent and do not benefit significantly from the prefix filters.

Bad samples generated by the LLMs are similarly centered around a few significant error modes, with the top 3 filters contributing to 63.2% of final-round testing filter triggers and the top 5 contributing to 76.9%, averaged across all seven models. The most active filters are generally QED filters targeting molecules with low QED scores, partly due to the fact that unconstrained Qwen-7B/14B and Gemma 12B are already highly likely (> 83.5%) to generate valid molecules. For these models, the prefix filters did not increase the validity rate statistically, while on Qwen-1.5B and Llama-8B (which have lower unconstrained validity rates) the filters did succeed in increasing validity.[5]

On the other hand, filters helped increase QED score with the exception of Qwen-1.5B, 14B, and Gemma4-31B. For Qwen-1.5B, and Gemma4-31B, the main gains for Qwen-1.5B are in validity rates, which do exhibit a significant increase. For Qwen-14B, mean QED only increased by 0.008 and was statistically insignificant. Unconstrained Qwen-14B already generates high-QED molecules averaging a score of 0.745 (on a scale of 0 to 1) with most samples being centered around the average; this makes it hard to learn prefix filters that clearly split the distribution into high- and low-QED molecules, and the learned filters were only able to prune a small number of molecules. On the other hand, tokens-per-sample overhead remained lower for molecules at a mean per-model increase of 25.4% (7.7 more tokens) across all seven models.

Like MLIR, filters learned for a specific model do not generally transfer well to other models. The one outlier is Qwen-14B's filters on Llama-8B: here, the filter "polycarbonyl-rich scaffolds" for Qwen has significant overlap with the top two filters for Llama in Panel A. The two Llama filters are unable to capture Qwen's error distribution, however: the Qwen polycarbonyl filter is a simple one-liner `smiles.count("C(=O)") >= 3` that triggers on a wide range of molecules, while the Llama filters are more specific.

E.3 Natural language summaries

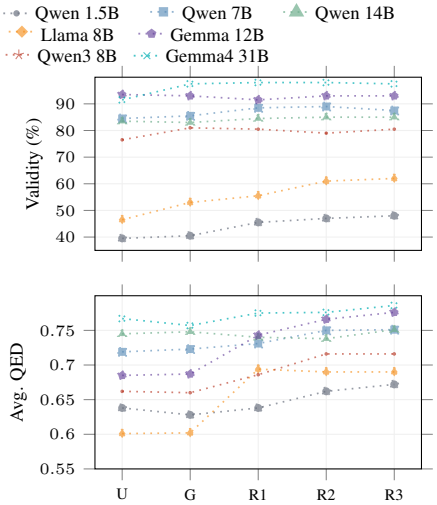
Figure 7 summarizes the experimental results for the natural-language summary benchmark, which again shows the same overall trends as MLIR and molecule generation. For this domain, we learn prefix filters that identify information leaks in the generated summaries. A summary is labeled safe

[5]The increase for Qwen-1.5B is insignificant at 95% bootstrap confidence intervals, and significant at 90%.

A. Most-triggered prefix filters (QED + Validity)

Model	Top triggered filters [learned round, type]
Qwen 1.5B	Seven-member aromatic carbocycle [r0, v] (18 events, 17.8%)
	Aryl amino / oxygenated low-QED motifs [r2, q] (10 events, 9.9%)
Qwen 7B	Saturated low-QED scaffold bundle [r1, q] (23 events, 27.7%)
	Rigid polar heterocycle / urea / lactam [r2, q] (19 events, 22.9%)
Qwen 14B	Polycyclic aryl acetate/ester scaffolds [r2, q] (24 events, 40.0%)
	Polycarbonyl-rich scaffolds [r0, q] (15 events, 25.0%)
Llama 8B	Large overdecorated polyacyl scaffolds [r0, q] (58 events, 37.7%)
	Large polyfunctional scaffolds [r0, q] (56 events, 36.4%)
Gemma 12B	Overgrown aryl ester without nitrogen [r0, q] (19 events, 23.5%)
	Overgrown polyaryl carbonyl/ether spine [r2, q] (18 events, 22.2%)
Qwen 3 8B	Bulky fused/bridged carbonyl scaffolds [r1, q] (32 events, 39.5%)
	Scaffolds with many carbonyl groups [r0, q] (14 events, 17.3%)
Gemma 4 31B	Cyclohexane-linked diamide motifs [r2, q] (4 events, 21.1%)
	Whitespace inside SMILES [r0, v] (3 events, 15.8%)

B. Molecule performance by method



C. Filter capture rates across models

	Q2.5 1.5B	Q2.5 7B	Q2.5 14B	L3.1 8B	G3 12B	Q3 8B	G4 31B
Q2.5 1.5B	46.3	8.5	8.6	7.4	6.8	8.1	1.8
Q2.5 7B	9.3	59.8	0.0	20.0	1.4	27.6	0.0
Q2.5 14B	16.7	12.2	44.4	38.5	17.8	50.4	1.8
L3.1 8B	11.7	6.1	9.9	65.2	1.4	23.6	20.0
G3 12B	7.4	26.8	43.2	14.1	79.5	53.7	0.0
Q3 8B	7.2	9.7	2.3	22.3	0.9	45.5	0.0
G4 31B	93.4	97.8	100.0	87.3	100.0	90.2	56.4

Figure 6: Summarized results for prefix filters over the molecule validity+QED setting. Panels report the same information as Figure 5, except that the second graph in panel B reports average QED scores instead of tokens per sample. `v` in panel A indicates that the filter is a validity filter, `q` indicates that it is a QED filter.

if it does not contain any leaks; the prefix filters result in a significant increase in the ratio of safe summaries, with the exception of Gemma3-12B, and Gemma4-31B, which already generate safe summaries 98.2% and 100% of the time.

Panel B of Figure 7 shows that the information count (the number of unprotected data values from the HR-MultiWoz dataset correctly included in the summary) per summary remained statistically stable as filters were added. In other words: the learned prefix filters successfully target just information leaks, without being overly aggressive and preventing useful information from appearing in the summaries.

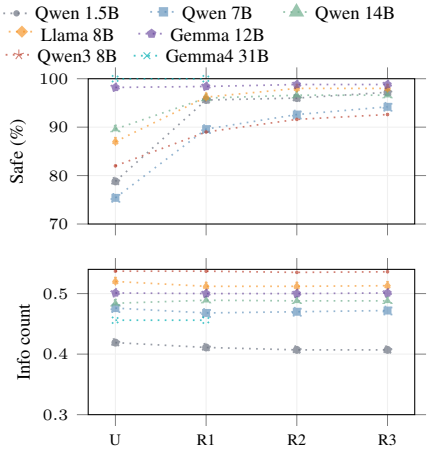
Tokens per sample overhead remained relatively low at a mean per-model increase of 27.0% (51.2 more tokens) across all seven models, using the final completed round for each model. Qwen-7B however was an outlier in token efficiency, incurring an overhead of 73.8% (105.6 more tokens per sample). This difference was due to the fact that Qwen-7B repeatedly tried to generate name-leaking summaries for a few specific tasks in the HR-MultiWOZ dataset; the biggest offender took 2139 tokens / sample to generate (compared to 143.1 tokens / sample unconstrained) and contributed 37.3 of the entire token count. In other words, the high token overhead for Qwen-7B is caused by a few pathological samples, and is not an overall trend.

Most failures in the natural language domain were concentrated around name leaks: only 36.8% of samples in unconstrained Qwen-1.5B with a leak contained a non-name leak; this metric was less

A. Most-triggered HR prefix filters per-model

Model	Top triggered filters [learned round]
Qwen 1.5B	Lead-ins: “name”, “named”, “identified as” [r0] (508 events, 34.7%)
	Lead-ins: “named”, “called”, “referred to” [r0] (176 events, 12.0%)
Qwen 7B	Direct name leaks after identity cues [r0] (804 events, 23.4%)
	Role-to-name leaks (e.g. “manager John”) [r0] (774 events, 22.5%)
Qwen 14B	Sentence-head leaks name (e.g. “John ...”) [r0] (238 events, 32.6%)
	Direct name leaks after naming cues [r0] (129 events, 17.7%)
Llama 8B	Attribution leaks (e.g. “approved by John”) [r0] (182 events, 24.9%)
	Role leaks (e.g., “the employee John”) [r0] (167 events, 22.8%)
Gemma 12B	Contact/coordinator/witness names [r0] (27 events, 55.1%)
	Training/contact coordinator names [r1] (8 events, 16.3%)
Qwen 3 8B	Names identifying witnesses or observers [r0] (407 events, 36.5%)
	Names in review or incident descriptions [r2] (171 events, 15.3%)
Gemma 4 31B	No filters learned; all baseline summaries are safe.

B. HR performance by method



C. Filter capture rates across models

	Q2.5 1.5B	Q2.5 7B	Q2.5 14B	L3.1 8B	G3 12B	Q3 8B	G4 31B
Q2.5 1.5B	95.3	85.4	80.8	87.7	44.4	95.6	N/A
Q2.5 7B	85.8	92.7	82.7	90.8	55.6	95.6	N/A
Q2.5 14B	69.8	81.3	82.7	81.5	55.6	67.8	N/A
L3.1 8B	87.7	77.2	67.3	90.8	33.3	95.6	N/A
G3 12B	59.4	48.8	44.2	43.1	55.6	64.4	N/A
Q3 8B	86.8	94.3	88.5	92.3	55.6	88.9	N/A
G4 31B	0.0	0.0	0.0	0.0	0.0	0.0	N/A

Gemma4-31B: no leaky baseline samples or learned filters.

Figure 7: Summarized results for prefix filters over the HR-MultiWOZ domain. Panels report the same information as in Figure 5, except that the second graph in Panel B plots displays the average number of leaks per sample instead.

than 25% for all other models. This error distribution is reflected in the top active filters in Figure 7, all of which are name-leak filters; an example filter here is:

```
def filter_name_named_or_identified_as(seq_prefix: str) -> bool:
    return bool(re.search(
        r'\b(?:named|called|identified as|referred to as)\s+'
        r'[A-Z][a-z]+(?:\s+[A-Z][a-z]+){0,2}\b',
        seq_prefix,
    ))
```

This filter simply identifies 4 common lead-ins to name leaks followed by a capitalized word (e.g., `called John`). Because all models share a similar failure mode, the summary domain is one where learned filters generalize relatively well across all models.

E.4 TypeScript

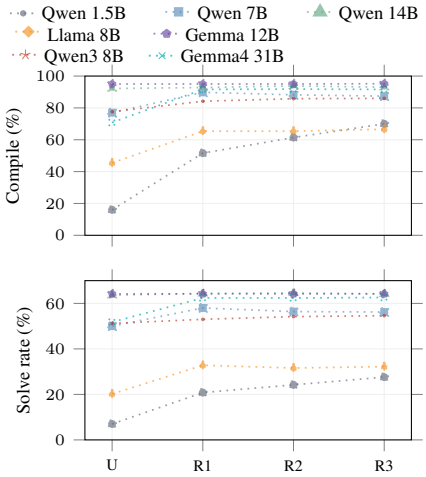
Figure 8 summarizes the results of the TypeScript domain, where we learn prefix filters that block programs containing errors that would prevent compilation. For the TypeScript domain, we reiterate that filters are learned from generating 5 samples each for 100 of the MultiPL-E TypeScript problems, then generating 10 samples each for a disjoint set of 50 MultiPL-E problems.

The filters succeed in significantly increasing compile rate, as evidenced on the top graph in Figure 8, with the exception of Gemma-12B and Qwen-14B, which alread generate compiling samples with roughly 95.0% accuracy. Increase in validity also translated into an increase in sample level solve

A. Most-triggered TypeScript prefix filters per-model

Model	Top triggered filters [learned round]
Qwen 1.5B	In-body ES `import` declarations [r0] (543 events, 15.8%)
	Stray `function` declaration fragments [r0] (491 events, 14.3%)
Qwen 7B	Impossible `if` conditions [r1] (168 events, 41.1%)
	Reassignment of `const` bindings [r0] (117 events, 28.6%)
Qwen 14B	Reassignment of `const` bindings [r0] (645 events, 77.4%)
	Block-scoped variables after block close [r2] (111 events, 13.3%)
Llama 8B	Use of `this` in standalone function bodies [r1] (789 events, 21.9%)
	Trailing stray `function` keyword [r0] (714 events, 19.8%)
Gemma 12B	Unfinished branches / control-flow cutoff [r0] (1 events, 33.3%)
	Late comment or call-stub cutoffs [r2] (1 events, 33.3%)
Qwen 3 8B	Updates to undeclared accumulators [r0] (56 events, 42.7%)
	Returns only inside nested loops [r0] (21 events, 16.0%)
Gemma 4 31B	Unmatched closing delimiters [r0] (146 events, 59.3%)
	Runs of identifiers without separators [r0] (34 events, 13.8%)

B. TypeScript performance by method



C. Filter capture rates across models

	Q2.5 1.5B	Q2.5 7B	Q2.5 14B	L3.1 8B	G3 12B	Q3 8B	G4 31B
Q2.5 1.5B	99.3	82.6	46.2	92.3	100.0	91.1	100.0
Q2.5 7B	79.0	71.3	71.8	70.7	44.0	93.8	82.2
Q2.5 14B	31.2	20.0	43.6	52.7	20.0	53.6	46.6
L3.1 8B	96.9	61.7	84.6	94.1	48.0	75.0	82.9
G3 12B	71.4	40.9	51.3	53.1	4.0	46.4	78.1
Q3 8B	91.2	67.0	46.2	70.3	12.0	55.4	78.1
G4 31B	16.0	13.0	59.0	63.4	20.0	89.3	82.2

Figure 8: Summarized results for prefix filters over the TypeScript domain. Panels report the same information as in Figure 5, except that Panel B shows compile rates (top) and per-sample solve rates (bottom), each computed over 500 test samples per model and method (10 samples for each of 50 tasks).

rates across all models,[6] indicating improvement on task performance. We do note that pass@10 was less improved: Qwen-1.5B was the only model showing a statistically significant increase for pass@10, from 13/50 tasks unconstrained to 29/50 tasks with filters. Our interpretation is that the domain-level filters learned by PALLA are effective when models possess the capability to solve a certain task (i.e., an answer exists within pass@10) but are hindered by simple syntactic or type errors, as evidenced by the increase in sample-level solve. However, prefix filters cannot let the model solve problems it cannot solve in the first place (i.e., none of the pass@10 answers are valid), which is expected, as the goal of prefix filters are to identify and prevent errors as opposed to introducing new information to the distribution of the LLM.

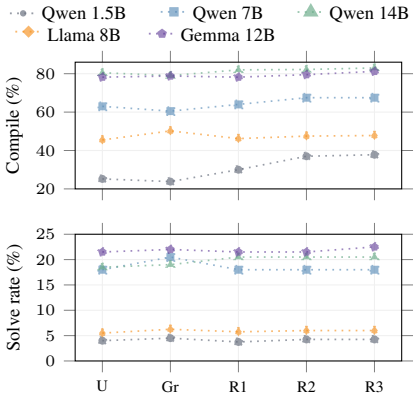
TypeScript is a domain where tokens per sample sometimes suffered significantly: Panel A in Figure 8 shows that the most active filters for Qwen-1.5B and Llama-8B were triggered over 1000 times. Overall, Qwen-1.5B went from requiring 46.1 tokens per sample unconstrained to 467.0 tokens per sample with all filters; Llama 8B went from 167.7 tokens per sample unconstrained to 838.8 tokens per sample with all filters (914.1% / 400.2% relative overhead respectively). Overhead was more modest for Qwen2.5-7B, Qwen2.5-14B, Qwen3-8B, and Gemma4-31B: 82.9%, 102.4%, 13.8%, and 27.8%, respectively. Gemma3-12B incurred essentially no overhead (0.04%): it had too few invalid examples to learn filters from, and the few filters that were learned barely fired.

[6]For Qwen3-8B, the model showed an increase on R1 but the difference became statistically insignificant in R3, due to Qwen3-8B learning filters that blocked previously compiling samples.

A. Most-triggered SQL prefix filters per-model

Model	Top triggered filters [learned round]
Qwen 1.5B	`AVG(...)` over nonnumeric columns [r1] (182 events, 15.8%)
	Aliased table later reused by original name [r0] (135 events, 11.7%)
Qwen 7B	Nonexistent columns [r1] (45 events, 58.4%)
	Formula-1 time strings cast to numeric types [r1] (16 events, 20.8%)
Qwen 14B	Mixed aggregate `SELECT` w/o `GROUP BY` [r0] (249 events, 65.9%)
	Scalar `races.name` subqueries [r1] (63 events, 16.7%)
Llama 8B	Duplicate top-level relation aliases [r1] (7 events, 28.0%)
	`LIKE` applied to non-text columns [r0] (5 events, 20.0%)
Gemma 12B	Numeric aggregates over text columns [r2] (16 events, 51.6%)
	Numeric columns compared to strs [r1] (13 events, 41.9%)

B. SQL performance by method



C. Filter capture rates across models

	Q2.5 1.5B	Q2.5 7B	Q2.5 14B	L3.1 8B	G3 12B
Q2.5 1.5B	74.9	39.2	47.4	56.4	40.2
Q2.5 7B	19.1	27.0	30.8	25.2	12.6
Q2.5 14B	25.8	17.6	46.2	23.4	41.4
L3.1 8B	14.4	2.0	3.8	7.3	6.9
G3 12B	17.1	2.0	11.5	11.5	24.1

Figure 9: Summarized results for prefix filters over the SQL domain. Panels report the same information as in Figure 5, except that Panel B shows sample-level oracle-valid rates (top) and per-sample solve rates (bottom), each computed over 400 test samples per model and method (10 samples for each of 40 tasks).

Filters for TypeScript generalize better than those for MLIR or molecules, as illustrated in Panel C of Figure 8. However, the TypeScript filters also show how filters from one model can be *distributionally unsound* (i.e., filter out even valid samples too often) on other models: for example, consider the fact that the Qwen-1.5B filters have a 100.0% fire rate on invalid samples from Gemma-12B. The invalid samples in Gemma are actually all caught by a single Qwen-1.5B filter:

```
def detect_unnecessary_whitespace(seq_prefix: str) -> bool:
    s = seq_prefix.strip()
    return True if (not s) else False
```

This filter detects whether the prefix is pure whitespace or padding, and rejects the current sample if so. In general, this filter is *unsound*: TypeScript functions can be compiled even if they start with a large number of spaces or newlines. However, this filter is *distributionally sound* for Qwen-1.5B: in our learning samples, all samples that start with whitespace eventually become invalid (e.g., by not generating a `return` statement for a function that returns a value), while all valid samples start with non-whitespace tokens. On the other hand, Gemma does not display this particular mode of behavior—it often generates valid samples that start with newlines. Thus this filter is distributionally unsound on Gemma, and ends up filtering many valid samples as well. The Qwen-1.5B-Gemma capture rate after removing this filter is still 56.0%., showing that TypeScript still does generalize much better than molecules or MLIR: most errors from LLMs are centered around the error codes TS1005 (syntax error—missing punctuation such as `;`), TS2355 (function not returning a return value), and TS2322 (assignment to a variable of incompatible type), which share filters.

E.5 SQL

Figure 9 summarizes the results of the SQL domain. For SQL, we only test over a representative set of five models: Qwen2.5-1.5B/7B/14B, Llama3 8B, and Gemma3-12B. SQL is a domain where compile rates do not show a statistically significant increase: only Qwen-1.5B, which increases from

25.2% unconstrained to 37.8% with all filters; the remaining models do exhibit a mild increase, but the increase is not statistically significant at the 95% bootstrap interval. Similarly, sample solve rates remain stable, even for Qwen-1.5B. The non-increase in solve rates are due to the fact that the models are generating more executable queries, but the queries do not lead to a correct answer.

Qwen-14B and Gemma-12B have around 80% success rates unconstrained; that their compile rate does not significantly increase is consistent with the rest of our observations that prefix filters struggle on improving performance for domains that LLMs are already competent at. Qwen-7B does show a mild increase from 63.0% compile unconstrained to 67.5% with all filters; the main problem with Qwen-7B was that a learning set size of 130 samples was too small with a compile rate of 63.0%, resulting in 48 invalid samples split across the 7 error buckets: the model did not have enough negative examples to learn generalizable prefix filters.

The outlier is Llama-8B, which failed to produce an increase in compile rate for a different reason: the model failed to learn prefix filters because many of the failure modes were non-prefix-catchable. For example, one of the filters proposed by QUERYLARGELLM was (simplified for presentation):

```
def filter_qualified_column_mismatch_on_bound_table(query_prefix: str) -> bool:
    q = query_prefix.lower()

    # Intended: catch references to non-existent columns, e.g. results.status
    if "from results" in q or "join results" in q:
        if re.search(r"\bresults\.(name|year|round|status)\b", q):
            return True

    # False-positive problem: `results.status` is bad, but `results.statusid` is valid
    return False
```

This filter is valid in principle: Llama-8B often fails by accessing the non-existent `results.status` column. However, there does exists `results.statusId` column inside the database, and thus a prefix filter targeting `results.status` becomes unsound, as it would trigger on valid queries containing `results.statusId` as well. These filters were thus rejected during the validation phase; consequently, PALLA could not learn filters that captured Llama-8B's most significant error modes, failing to improve performance. This case illustrates a failure mode of PALLA discussed in Section 6: when error modes are not soundly capturable by prefix filters.

Filter transfer rates for SQL remain low; the models display different patterns of errors.

F GPT-OSS-120B AS AN ALTERNATIVE FILTER SYNTHESIZER

We run the entire PALLA pipeline on a reduced representative model set of Qwen-2.5B/7B/14B, Llama3-8B, and Gemma3-12B (on all four main-text domains), while using the open-source model GPT-OSS-120B as the synthesizer, to illustrate that our results are not sensitive to the choice of external synthesizer. Figure 10 reports validity and token costs in the style of Figure 3 in the main text, while Table 1 compares the final-stage results of using GPT-OSS-120B and GPT5.4-mini.

Overall, GPT-OSS-120B showed very similar trends to GPT-5.4-mini, showing significant increases for domain-model pairs where model performance was not saturated. The filters produced by GPT often produced slightly better validity rates after all three rounds of filter learning, although the difference was small. On the other hand, the tokens-per-sample efficiency was slightly better for GPT-OSS-120B, although again the difference was small. Filter learning using GPT-OSS-120B took 54.7 GPU hours for all three rounds of filter learning on NVIDIA LS40 GPUs, similar to the QLoRA baseline in Section 5.4, and 16.9 GPU hours for the first round of filter learning.

G GRAMMAR-CONSTRAINED GENERATION WITH PALLA

In this section, we show that PALLA can be deployed in tandem with grammar-constrained decoding, to target and filter errors that grammar-constrained decoding alone cannot. We illustrate the application by rerunning the MLIR domain on Qwen2.5-1.5B/7B/14B, Llama3-8B, and Gemma3-12B models, collecting all samples for filter learning while grammar-constrained decoding is enabled.

Table 2 illustrates the results; PALLA produces a clear increase in validity rate over all five models even when grammar-constrained decoding is active. This outcome is expected, as discussed in

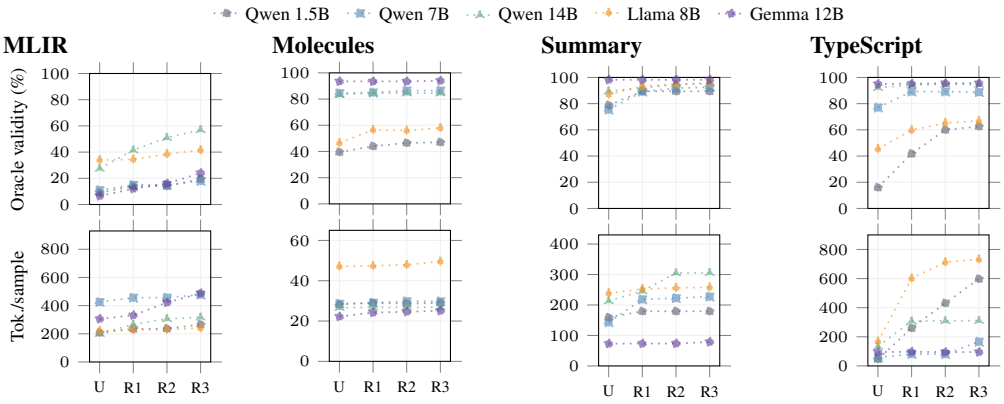


Figure 10: Validity and token costs per sample using GPT-OSS-120B as the filter synthesizer. The layout, model colors, and axis scales match Figure 3. U denotes unconstrained generation; R1–R3 denote generation with the successive OSS filter banks. The top row reports oracle validity, and the bottom row reports tokens per sample. Molecules use the joint validity+QED setting.

Table 1: Final-round comparison of GPT-OSS-120B (OSS) and GPT-5.4-mini (GPT) as filter synthesizers. U reports unconstrained validity; OSS and GPT report R3 results. Higher oracle validity and lower tokens per sample are preferable; bold identifies the better value within each OSS/GPT pair. Molecules use the joint validity+QED setting.

		Oracle validity (%)			Tokens/sample	
Domain	Model	U	OSS	GPT	OSS	GPT
MLIR	Qwen 1.5B	9.0	19.5	**31.5**	**266.4**	325.8
	Qwen 7B	11.0	18.0	**43.0**	**477.9**	700.3
	Qwen 14B	27.5	57.0	**71.0**	**317.5**	358.4
	Llama 8B	34.0	41.5	**49.0**	**239.8**	289.0
	Gemma 12B	6.5	**24.0**	22.5	**487.6**	523.0
Molecules	Qwen 1.5B	39.5	47.0	**48.0**	**29.0**	36.2
	Qwen 7B	84.5	86.5	**87.5**	**29.8**	40.3
	Qwen 14B	83.5	84.5	**85.0**	**26.8**	33.5
	Llama 8B	46.5	58.0	**62.0**	**49.6**	58.4
	Gemma 12B	93.5	**94.0**	93.0	**25.1**	30.7
Summary	Qwen 1.5B	78.8	89.4	**97.2**	**179.3**	203.6
	Qwen 7B	75.4	92.6	**94.2**	**227.4**	248.7
	Qwen 14B	89.6	95.2	**96.6**	306.1	**244.0**
	Llama 8B	87.0	95.8	**98.0**	**257.9**	269.3
	Gemma 12B	98.2	98.2	**98.8**	78.6	**75.6**
TypeScript	Qwen 1.5B	16.0	62.6	**70.0**	597.7	**467.0**
	Qwen 7B	77.0	**88.8**	87.4	166.2	**107.6**
	Qwen 14B	92.2	**94.6**	93.0	311.3	**270.6**
	Llama 8B	45.4	**66.8**	66.6	**731.8**	838.8
	Gemma 12B	95.0	**95.6**	95.2	**95.0**	95.3

Section 5.1: in MLIR, most major errors from LLMs were errors that were not preventable by the deployed MLIR grammar (or grammars in general), such as the use of custom operators. In particular, we also observe that grammar-constrained decoding and PALLA had an overall synergistic approach: on 4 out of 5 models, grammar constrained decoding + PALLA outperformed PALLA-only learned filters in terms of validity rate, with a high rate of difference as well. We are unsure of why Qwen-1.5B does not display this synergistic behavior, but conjecture that it may be due to the fact that the grammar filters out enough samples such that PALLA cannot learn a set of highly effective filters.

Table 2: MLIR compile rates (%). Grammar R1–R3 progressively learn and apply prefix filters over grammar-constrained generations. Paper R3 reports the original final-round results without grammar-constrained decoding. Bold marks the highest compile rate in each row.

Model	Grammar	Grammar R1	Grammar R2	Grammar R3	Paper R3
Qwen 1.5B	5.0	5.5	13.0	14.5	**31.5**
Qwen 7B	26.0	54.5	63.0	**68.0**	43.0
Qwen 14B	17.5	55.5	67.5	**72.5**	71.0
Llama 8B	25.0	47.5	55.5	**59.5**	49.0
Gemma 12B	11.0	12.5	21.0	**31.5**	22.5

Domain		Qwen 2.5 1.5B	Qwen 2.5 7B	Qwen 2.5 14B	Llama 3.1 8B	Gemma 3 12B	Qwen 3 8B	Gemma 4 31B	Own	Cross	Ratio
MLIR	Full	74.7% [68.1,80.8]	74.2% [67.4,80.3]	84.8% [78.6,90.3]	49.2% [40.9,57.6]	64.2% [57.2,71.1]	88.1% [83.1,92.7]	64.9% [56.1,73.7]	71.5% [68.9,74.0]	27.7% [26.0,29.6]	0.39 [0.36,0.41]
(catch %)	Top-3	33.5% [26.9,40.1]	51.7% [44.4,59.0]	40.0% [32.4,48.3]	22.0% [15.2,28.8]	27.3% [20.9,33.7]	65.5% [58.2,72.3]	51.8% [43.0,60.5]			
Molec.	Full	44.9% [37.7,52.7]	58.1% [48.4,67.7]	46.5% [36.0,57.0]	59.6% [51.8,66.9]	74.3% [66.1,82.6]	45.5% [36.6,54.5]	56.4% [43.6,69.1]	55.0% [51.4,58.7]	25.6% [24.6,26.7]	0.47 [0.43,0.50]
(catch %)	Top-3	16.8% [11.4,22.8]	35.5% [25.8,45.2]	38.4% [27.9,48.8]	48.2% [40.4,56.0]	62.4% [53.2,71.6]	38.2% [30.1,47.2]	43.6% [30.9,56.4]			
Summ.	Full	95.3% [90.6,99.1]	92.7% [87.8,96.7]	82.7% [71.2,92.3]	90.8% [83.1,96.9]	55.6% [22.2,88.9]	88.9% [82.2,94.4]	N/A	84.3% [78.3,90.1]	73.3% [69.1,77.6]	0.87 [0.81,0.94]
(catch %)	Top-3	75.5% [67.0,84.0]	48.8% [39.8,57.7]	42.3% [28.8,55.8]	32.3% [21.5,44.6]	33.3% [11.1,66.7]	53.3% [43.3,63.3]	N/A			
TS	Full	99.3% [98.3,100.0]	71.3% [62.6,79.1]	74.4% [59.0,87.2]	94.1% [91.2,96.7]	4.0% [0.0,12.0]	55.4% [46.4,64.3]	82.2% [76.0,88.4]	68.7% [65.6,71.7]	63.8% [61.7,65.9]	0.93 [0.89,0.97]
(catch %)	Top-3	75.2% [71.0,79.3]	16.5% [10.4,23.5]	35.9% [20.5,51.3]	57.5% [51.6,63.4]	0.0% [0.0,0.0]	30.4% [21.4,39.3]	79.5% [72.6,85.6]			
SQL	Full	75.3% [70.2,79.9]	27.0% [20.3,34.5]	46.2% [34.6,56.4]	8.7% [5.0,12.8]	24.1% [16.1,33.3]	–	–	36.3% [32.9,39.7]	22.8% [20.9,24.8]	0.63 [0.57,0.70]
(catch %)	Top-3	32.1% [27.1,37.5]	24.3% [17.6,31.1]	41.0% [30.8,52.6]	5.0% [2.3,8.3]	24.1% [14.9,33.3]	–	–			

N/A: no invalid Gemma 4 summaries; –: not evaluated. SQL covers five models. Panel C excludes empty source banks and undefined rates.

Figure 11: Panels B (left) and C (right) from Figure 2 annotated with 95% bootstrap confidence intervals, represented as the bracketed intervals on the bottom. Noise is centered on cases where unconstrained models are already competent and the number of samples available is low (e.g., Summary & Gemma-12B, or TypeScript & Gemma-12B).

H FILTER COVERAGE STATISTICS WITH INTERVALS

Figure 11 presents a version of panels B and C from Figure 2 with 95% bootstrap confidence intervals.